VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Systém předávání zpráv na platformě J2EE

Message Sending System in the J2EE Platform

2010

Bc. Aleš Haladej

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2010 .....

Abstrakt

Hlavním záměrem této práce je popsat principy a využití systémů pro předávání zpráv (messaging systémy). Důraz je kladen především na možnosti použití na platformě Java Enterprise Edition prostřednictvím technologie Java Message Service. Úvodní část práce se zabývá obecnými principy systémů předávajících si zprávy spolu s aspekty asynchronní komunikace mezi systémy (včetně představení jejích účastníků). V další části práce seznamuje čtenáře se specifikací technologie Java Message Service a jejím aplikačním rozhraním. Následně jsou stručně představeny některé existující implementace messaging systémů, přičemž je blíže popsána open-source implementace HornetQ (dříve JBoss Messaging) od divize JBoss společnosti Red Hat, Inc. a také možnost napojení na aplikace jiných platforem prostřednictvím protokolu STOMP. Na závěr práce je přiblížena implementace ukázkové aplikace, demonstrující rozličné schopnosti Java Message Service API.

Klíčová slova: asynchronní komunikace, distribuovaný systém, HornetQ, Java Enterprise Edition, Java Message Service, Message-Oriented Middleware, předávání zpráv

Abstract

The main intention of this thesis is to describe the principles and the utilization of message sending systems. The stress is mainly laid on the options of application on the Java Platform, Enerprise Edition by means of the Java Message Service technology. The opening part of the thesis deals with the general principles of messaging systems along with the aspects of asynchronous communication between systems (including an introduction of its participants). In the following part readers are apprised of the contents of the Java Message Service Specification and its application programming interface. Subsequently some existing implementations of messaging systems are briefly introduced, whereas the open-source implementation of HornetQ (formerly known as JBoss Mesaging) from JBoss, a division of Red Hat, Inc., is described more closely as well as the option to connect it to other platforms' applications through the STOMP protocol. At the end of the thesis an implementation of the example application is outlined, showing the various features of the Java Message Service API.

Keywords: asynchronous communication, distributed system, HornetQ, Java Enterprise Edition, Java Message Service, Message-Oriented Middleware, message passing

Seznam použitých zkratek a symbolů

ACL	–	Access Control List
AIO	–	Asynchronous Input/Output
AMQP	–	Advanced Message Queuing Protocol
API	–	Application Programming Interface
BMT	–	Bean-Managed Transactions
CMT	–	Container-Managed Transactions
CORBA	–	Common Object Request Broker Architecture
DAO	–	Data Access Object
DCOM	–	Distributed Component Object Model
EAI	–	Enterprise Application Integration
EJB	–	Enterprise JavaBean
ESB	–	Enterprise Service Bus
FTP	–	File Transfer Protocol
HTTP	–	HyperText Transfer Protocol
HTTPS	–	HyperText Transfer Protocol Secured
IP	–	Internet Protocol
J2EE	–	Java 2 Platform, Enterprise Edition
J2SE	–	Java 2 Platform, Standard Edition
JAAS	–	Java Authentication and Authorization Service
Java EE	–	Java Platform, Enterprise Edition
JCA	–	Java EE Conector Architecture
JCP	–	Java Community Process
JDBC	–	Java DataBase Connectivity
JMS	–	Java Message Service
JNDI	–	Java Naming and Directory Interface
JRE	–	Java Runtime Environment
JSP	–	JavaServer Pages
JSR	–	Java Specification Request
JTA	–	Java Transaction API
JTS	–	Java Transaction Service
JVM	–	Java Virtual Machine
LDAP	–	Lightweight Directory Access Protocol
MDB	–	Message Driven Bean
MOM	–	Message-Oriented Middleware
NMS	–	.NET Message Service
OTP	–	Open Telecom Platform
PHP	–	Personal Home Page
POJO	–	Plain Old Java Object

REST	–	Representional State Transfer
RMI	–	Remote Method Invocation
RPC	–	Remote Procedure Call
SOA	–	Service Oriented Architecture
SOAP	–	Simple Object Access Protocol
SQL	–	Structured Query Language
SSL	–	Secure Sockets Layer
STOMP	–	Streaming Text Orientated Messaging Protocol
TCP	–	Transmission Control Protocol
TLS	–	Transport Layer Security
UDDI	–	Universal Description, Discovery and Integration
UML	–	Unified Modeling Language
URI	–	Uniform Resource Identifier
VPN	–	Virtual Private Network
W3C	–	World Wide Web Consortium
WAN	–	Wide Area Network
WSDL	–	Web Service Description Language
XML	–	eXtensible Markup Language

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Podnikové systémy pro zasílání zpráv (neboli messaging systémy) se stávají základní stavební komponentou pro integraci interních procesů obchodních společností. Tyto produkty mnohdy označované jako middleware založený na předávání zpráv neboli *Message-Oriented Middleware* (zkráceně *MOM*) umožňují, aby oddělené systémové komponenty a aplikace mohli být zkombinovány do spolehlivého a přitom pružného systému, čímž výrazně usnadňují tvorbu distribuovaných aplikací napříč heterogenními platformami.

Aplikace a služby napsané v jazyce Java by měli být schopné používat tyto messaging systémy. Proto byla navržena a vyvinuta specifikace *Java Message Service* (ve zkratce *JMS*), která na platformě *Java, Enterprise Edition*, (*Java EE*) poskytuje jednotný přístup k těmto systémům při zachování přenositelnosti vyvinutých komponent.[1]

Tato práce si klade za cíl uvést čtenáře do problematiky systémů předávání zpráv, seznámit jej s principy tvorby distribuovaných systémů prostřednictvím middleware produktů na platformě Java EE s využitím JMS a případy použití takových produktů, představit možnosti a sílu JMS stejně tak, jako úskalí a nástrahy, jenž s sebou přináší jeho integrace. Čtenář bude blíže obeznámen s vlastnostmi a specifiky produktu HornetQ, jenž je využit pro implementaci demonstračních příkladů, v nichž budou názorně předvedeny široké možnosti JMS. Dojde také na představení protokolu STOMP, jehož pomocí si lze předávat zprávy s HornetQ serverem i mimo platformu Java, tedy s využitím klientů vytvořených v jiných programovacích jazycích. Tuto diplomovou práci bude tedy možno do jisté míry používat jako příručku, či manuál při zájmu o některou z uvedených technologií a problematik.

1.1 Požadavky a předpoklady

Pro zvládnutí textu v plné míře se předpokládá čtenářova základní znalost některých pojmů z problematiky distribuovaných systémů, síťových protokolů a především pak principů programovacího jazyka Java™ a platformy Java EE, včetně několika základních technologií této platformy jako jsou JDBC, RMI, JNDI, EJB, servlety a JSP. Vhodné je také povědomí o konceptech tvorby XML dokumentů a fungování webových služeb spolu s technologiemi, na nichž jsou postaveny.

1.2 Struktura práce

Po úvodní kapitole práce pokračuje kapitolami, které čtenářům přibližují pozadí problematiky tvorby distribuovaných systémů za pomoci produktů pro předávání zpráv spolu s principy a termíny, jenž se v této doméně objevují. Následně jsou stručně představeny některé protokoly, které se pro předávání zpráv používají, přičemž je podrobněji rozebrán protokol STOMP.

Technologie JMS je podrobně popsána až v následující obsáhlé kapitole. Její podkapitoly prezentují motivaci jejího vzniku, její architekturu a strukturu jejích jednotlivých

konceptů a komponent, jimiž je tvořena. Představena jsou také nejpoužívanější rozhraní jejího aplikačního rozhraní včetně způsobu jejich použití.

Poté (v kapitole 5) jsou zmíněny také otázky zabezpečení těchto systémů a způsob ochrany dat.

Následující kapitola se zabývá open source messaging systémem HornetQ, který byl použit pro tvorbu demonstračního příkladu. Jsou představeny jeho vlastnosti, možnosti a specifika.

Na závěr je stručně představena implementace demonstračního příkladu s využitím JMS API, protokolu STOMP a HornetQ jako messaging serveru.

2 Předávání zpráv v distribuovaném prostředí

S nástupem internetu se pro společnosti usilující o tvorbu pružných a škálovatelných podnikových aplikací stalo mnohem důležitější využití distribuovaného zpracování úloh. Označení distribuovaný systém (*distributed system*) předznamenává, že různé části systému mohou být umístěny na různých strojích. Tyto stroje mohou ležet vedle sebe ve stejné místnosti či mohou být rozmístěny v různých zemích po celém světě. Stroje se však umisťují tam, kde jsou potřeba, a různé části distribuovaného systému běží na strojích, které se pro danou oblast systému nejvíce hodí.

Při dekompozici složité aplikace na několik komponent a při jejich následné instalaci na více strojů je nutno počítat s mnoha dalšími faktory, které ovlivňují návrh systému, jako jsou nesourodá architektura strojů (např.: Intel vs. Alpha), rozdílné operační systémy, šířka pásma sítě (tj. množství dat, jež lze přenést od jednoho stroje k jinému, potažmo rychlost přenosu dat) a mnoho dalších důvodů, kvůli nimž může síťová komunikace havarovat. Ve zkratce, složitost distribuovaného systému je exponenciálně vyšší, než složitost ekvivalentního systému nedistribuovaného.[2]

2.1 Dekompozice distribuovaného systému

Samotný distribuovaný systém (resp. jeho kód) může být logicky rozložen nejméně na dva díly — vlastní *funkční kód* a *kód infrastruktury*.

2.1.1 Funkční kód

Funkční kód zajišťuje vlastní byznys funkcionalitu systému a nezávisí na skutečnosti, zda je systém distribuovaný či nikoliv. Funkční kód je tedy reprezentován komponentami, moduly, aplikacemi, funkcemi a procedurami, jimiž je implementována samotná agenda systému.

2.1.2 Kód infrastruktury

Na druhou stranu kód infrastruktury je přímo podmíněn distribuovaností systému. Jestliže systém není distribuovaný, tato porce kódu téměř vymizí. Naproti tomu v distribuovaném systému může být kód infrastruktury velmi složitý a může být dokonce rozsáhlejší než vlastní funkční kód. Primárním účelem kódu infrastruktury je přenos dat mezi jednotlivými částmi distribuovaného systému, tj. zajištění komunikace a spolupráce modulů systému.

Takový kód sice neplní žádnou byznys agendu systému a nepřináší tak koncovému uživateli žádnou hodnotu, avšak z pochopitelných důvodů jej nelze z aplikace zcela eliminovat. Naštěstí kód infrastruktury není závislý na byznys účelu systému, nýbrž pouze na jeho distribuovanosti, a může být tedy použit opakovaně při podobné situaci v jiných systémech. Proto nejlepší východisko z této situace nabízí znovupoužitelnost.

2.2 Middleware

Bylo vyvinuto mnoho knihoven, které položily základ pro vytvoření řady standardů tvorby distribuovaných systémů. K nejrozšířenějším patří technologie *Distributed Component Object Model* (*DCOM*) od společnosti Microsoft, *Remote Method Invocation* (*RMI*) od firmy Sun či *Common Object Request Broker Architecture* (*CORBA*) skupiny OMG. Tento kód infrastruktury je běžně označován jako *middleware*, neboť vystupuje jako jakýsi prostředník (mezivrstva) mezi komunikujícími aplikacemi. Jedná se tedy o software používaný pro vzájemné propojení softwarových aplikací, poskytující specializované služby a interoperabilitu mezi distribuovanými aplikacemi.[3]

Na základě přístupu, jehož middleware používá pro přenos dat mezi distribuovanými softwarovými aplikacemi, můžeme rozlišit dva podstatně rozdílné typy middleware. Jedná se o middleware založený na vzdáleném volání procedur (*RPC-based middleware*) a middleware založený na zasílání zpráv (*Message-oriented middleware*).[1]

2.3 Vzdálené volání procedur

Do kategorie middleware založeného na vzdáleném volání procedur, kam pro tentokrát zařadíme i software pro dotazování objektů a volání jejich metod, je postaven, jak název napovídá, na koncepci vzdáleného volání procedur, neboli *Remote Procedure Call* (*RPC*).

Vzdálené volání procedur (řídčeji označováno termínem vzdálené volání funkcí) je typ meziprocesní komunikace, která umožňuje spouštět procedury vzdáleného systému (tzv. *serveru*), tedy obecně nacházející se v jiném adresním prostoru (v běžném případě umístěném na jiném stroji dostupném prostřednictvím sdílené sítě), aniž by programátor musel explicitně implementovat podrobnosti této vzdálené interakce. Výsledky vzdáleného volání jsou přitom přeneseny do adresního prostoru volající aplikace (označované termínem *klient*), kde mohou být použity k dalšímu zpracování. Důsledkem tohoto principu je situace, kdy programátor víceméně píše stejný kód, ať již je volaná procedura lokální, či vzdálená.

2.3.1 Průběh vzdáleného volání

Vzdálené volání procedury (resp. funkce) nastává, když klient vyšle požadavek vzdálenému serveru, aby spustil určitou proceduru (funkci) s danými parametry. Předem známý vzdálený server spustí požadovanou proceduru a po ukončení jejího běhu zašle klientovi odpověď a klientská aplikace pokračuje dál ve svém zpracování. Zatímco server zpracovává volání, klient je blokován a čeká, dokud server nedokončí zpracovávání a nevrátí mu odpověď. Teprve až poté obnoví svůj běh.

Příkladem podobného principu z reálného světa může být telefonní hovor mezi dvěmi osobami. Jedna osoba začne mluvit k druhé, zatímco ta poslouchá, dokud první neskončí.

[1] Pojem *middleware* nepokrývá pouze dvě zmiňované kategorie produktů, ale i produkty pro přístup k datům (tzv. databázový, nebo také na SQL orientovaný middleware), pro dotazování objektů (Object Request Brokers (ORB)), pro monitorování zpracování distribuovaných transakcí (Distributed transaction processing (DTP) monitors) a další.

Druhá osoba zpracuje získané informace a odpoví první osobě. Po celou tuto dobu první osoba trpělivě čeká na odpověď druhé osoby.

Jako zástupce tohoto typu chování lze uvést technologie Java RMI, CORBA, Microsoft DCOM, ale třeba i webové služby.

Architektura takovéhoto systému bývá podle účinkujících označována termínem *klient-server*, přičemž na model samotné komunikace je poukazováno jako na princip *požadavek-odpověď* (*request-reply*). Navíc vzhledem k tomu, že klientská aplikace, používající pro přenos dat middleware založený na vzdáleném volání procedur, musí čekat, než aplikace serveru ukončí zpracovávání dat, označujeme tento typ komunikace jako *synchronní* a o komunikujících procesech hovoříme jako o těsně svázaných (*tightly coupled*) procesech jeden k druhému.[4]

2.3.2 Výkon systémů založených na RPC

Výkon distribuovaných systémů používajících jako komunikačního prostředku vzdálené volání procedur je tedy vzhledem k nutnosti čekat na vyřízení jednoho požadavku před zpracováním dalšího limitován především dobou zpoždění na síti vznikajícím přenosem dat z klienta na server a zpět (tzv. *round-trip time*) neboli latencí (*latency*) sítě.[5]

2.4 Middleware založený na zasílání zpráv

Middleware založený na posílání zpráv neboli *Message-Oriented Middleware* (zkráceně *MOM*)[2] představuje zcela odlišný typ middleware, jehož názorný příklad může představovat komunikace dvou osob prostřednictvím poštovní služby. První osoba nečeká po odeslání zásilky na odpověď, než začne dělat cokoliv jiného, ale dál normálně pokračuje ve své činnosti. Odpověď od druhé osoby obdrží až v nějakém okamžiku poté, aniž by činnosti prováděné v mezidobí mezi těmito událostmi na obdržené odpovědi nějak závisely. Navíc pokud není osoba adresáta zásilky zrovna dostupná v době doručení, je zásilka uchována (na poště, či ve schránce) do doby, než si ji může vyzvednout.

2.4.1 Principy MOM

Myšlenka, na níž staví messaging systémy, je tedy v celku jednoduchá. Spočívá totiž v umístění prostředníka mezi libovolné dvě části distribuovaného systému, jež potřebují navzájem komunikovat (a tedy přenášet data). Místo aby byla data přenášena přímo od jedné aplikace k druhé, přenášejí aplikace tato data přes prostředníka. Tímto prostředníkem je právě messaging systém (v předchozím příkladu zastupován poštovní službou), který tak odstraňuje vazbu mezi komunikujícími aplikacemi, neboť aplikace komunikují vždy pouze s ním a nikdy přímo. Aplikace tedy vůbec ani nemusejí vědět o své existenci. Tento princip bývá označován jako *asynchronní* komunikace[3], přičemž hovoříme o volné vazbě mezi aplikacemi (tzv. *loosely coupled* aplikace).[6] Komunikující

[2] Systémy MOM jsou pro jednoduchost v textu dále označovány jako messaging systémy.

[3] Většina messaging systémů sice také podporuje jistou formu synchronní komunikace typu požadavek-odpověď, ale tato schopnost není těžištěm těchto systémů.

aplikace (označovány jako *klienti*) mají navzájem rovnocenné postavení. Mluvíme o tzv. *klient-klient* architektuře (neboli *peer-to-peer*).

Dalším aspektem messaging systémů, jenž zasluhuje pozornost, je fakt, že v případě, kdy druhá aplikace je z nějakého důvodu nedostupná (kupř. aplikace neběží nebo se vyskytl síťový problém), MOM zajistí, aby žádná aktuálně nedoručitelná zpráva nebyla ztracena, a pozdrží její odeslání do doby, než bude její doručení možné. Tohoto efektu je většinou dosaženo pomocí uchovávání zpráv ve zvláštním úložišti označovaném jako *fronta zpráv*. Tento mechanismus umožňuje aplikacím distribuovaného systému používajícím MOM, aby měly zcela odlišné doby běhu, čímž je potlačena potřeba souběžného chodu aplikací. Messaging systém se navíc stará, nejen aby zprávy nebyly ztraceny, ale ani doručeny mimo pořadí či duplikovány.[2]

Tato uvolněná koncepce umožňuje, aby moduly systému byly distribuovány napříč heterogenním prostředím (přesahujícím různé platformy, operační systémy a síťové protokoly), a zároveň snižuje náročnost vývoje aplikací (izolováním vývojáře od detailů jednotlivých operačních systémů a síťových rozhraní).

2.4.2 Výkon MOM

Výkon asynchronních systémů, kdy lze rozvádět tok zpráv různými směry, aniž by bylo nutné čekat na jejich zpracování, jako je tomu u systémů synchronních, není limitován latencí sítě, avšak její propustností (*bandwidth*). To vývojářům dovoluje vytvářet většinou daleko výkonnější aplikace.[5]

2.4.3 Výhody MOM

Jak již bylo možno vypozorovat v předchozích pasážích textu, systémy interagující asynchronním způsobem mohou snížit či zcela odstranit některá rizika spojená se synchronní interakcí. Hlavní výhody protokolů asynchronní komunikace založené na předávání zpráv leží především v jejich schopnosti uchovávat, směrovat a transformovat zprávy v procesu jejich doručování.[6]

Neboť požadavky ve formě zpráv jsou uchovány ve frontě zpráv, dokud si je příjemce nevyzvedne, není nutné, aby obě strany komunikace – odesílatel a příjemce – byly dostupné na síti ve stejnou dobu. Zároveň jsou tak sníženy celková režie komunikace včetně požadavků na systémové zdroje (udržování aktivního spojení a relace je dosti nákladné) a nebezpečí selhání. Komunikace jednotlivých částí systému prostřednictvím front zpráv tedy nabízí větší míru spolehlivosti a flexibility.

Další výhodou messaging systémů je schopnost směrovat zprávy uvnitř samotné messaging mezivrstvy. Toto je dovedeno dokonce ještě o krok dále, neboť MOM umožňuje, aby jedna zpráva byla doručena více příjemcům. MOM navíc většinou poskytuje možnosti pro transformaci zpráv takovým způsobem, aby jejich formát vyhovoval potřebám komunikujících klientů.

2.4.4 Nevýhody MOM

Ovšem nic nemá pouze světlé stránky a ani MOM není výjimkou. Primární nevýhodou systémů využívajících MOM je především fakt, že zasazují do architektury další komponentu v podobě vlastního messaging systému (někdy označovaného jako *message broker*). Toto může obecně vést ke snížení výkonnosti (zpomalení) a spolehlivosti systému jako celku a také to může zapříčinit zvýšení obtížnosti a nákladů na údržbu systému. Více o vlivech využití MOM na architekturu systému viz kapitola 3.2.

Nevýhodu lze také spatřovat v tom, že messaging systémy nelze vždy využít pro zajištění veškeré komunikace mezi aplikacemi, neboť asynchronní povaha interakce nevyhovuje všem situacím. Ačkoliv většina MOM produktů obsahuje prostředky pro zajištění synchronní komunikace, tak ani tyto schopnosti nepostihují všechny případy.

Bohužel výrazným nedostatkem, za nějž však tvůrci messaging systémů přímo nemohou, je absence standardů v oblasti middleware pro předávání zpráv. Každá větší SW společnost s vlastní implementací MOM poskytuje vlastní komunikační protokol, aplikační rozhraní a sadu nástrojů pro práci a správu jejich produktu.

Tuto situaci do jisté míry řeší a zachraňuje na platformě Java EE specifikace standardu Java Message Service a jeho aplikační rozhraní, která získala na velké popularitě, a je tedy naštěstí implementován většinou MOM produktů (výjimkou je třeba MSMQ firmy Microsoft). JMS je však pouze sadou rozhraní s definovaným významem, která sjednocují způsob, jakým JMS klienti využívají prostředky messaging systémů. JMS však nedefinuje formát zpráv, protokol komunikace, ani způsob implementace messaging systémů, takže různé JMS systémy nejsou schopny vzájemně spolupracovat. JMS nabízí pouze přenositelnost vyvinutých komponent a aplikací v rámci Java EE platformy, tj. schopnost jednou vyvinutých komponent pracovat s libovolným MOM produktem kompatibilním s JMS, a tudíž možnost jejich výměny.

Rovněž situace na poli protokolů pro výměnu zpráv mezi systémy není zcela růžová. Jak bylo již řečeno, každý MOM produkt definuje vlastní protokol, vyhovující jeho potřebám, a neřeší možnost spolupráce s jinými produkty. Oblibu sice získává standard *Advanced Message Queuing Protocol* (*AMQP*), který již podporuje mnoho messaging systémů, ale jeho rozšíření není zatím dostatečné, aby se stal standardem pro celou doménu systémů pro předávání zpráv. AMQP a další protokoly jsou představeny v kapitole 3.5.

2.4.5 Enterprise Service Bus

Velké podniky často používají messaging systémy pro implementaci sběrnice zpráv, jejíž prostřednictvím volně svazují dohromady heterogenní systémy, což vede k vytvoření pružnější a škálovatelnější architektury, neboť vzhledem k neexistenci těsných vazeb mezi propojovanými systémy lze do architektury snáze přidávat nové systémy a vyřazovat ty zastaralé. Tyto sběrnice tak tvoří architekturu, či softwarovou infrastrukturu známou jako *Enterprise Service Bus* (*ESB*).

Messaging systém tedy vystupuje jako událostmi řízený (*event-driven*) a na standardech založený pohon ESB, která nad ním poskytuje abstrakci, jež umožňuje podnikové služby snadno a volně propojovat. ESB software se snaží nahradit veškerou přímou

komunikaci aplikací komunikací prostřednictvím sběrnice. ESB musí tedy zapouzdřovat smysluplným způsobem funkcionalitu jejích komponent poskytováním rozhraní ve formě standardní sady zpráv, které přijímá a odesílá.[7]

2.5 Webové služby

Webová služba představuje aplikační rozhraní umístěné na webu (tzn. přístupné prostřednictvím protokolu HTTP) a spouštěné na vzdálených strojích, kde webové služby běží. Důležitou vlastností webových služeb je jejich naprostá nezávislost na platformě (tj. operačním systému, programovacím jazyku apod.). Neboť jsou založeny na otevřených standardech a protokolech zaštítěných konzorciem W3C, jsou relativně jednoduché a je vcelku snadné je implementovat. Proto také většina dnešních platforem nabízí podporu pro jejich tvorbu a prostředí pro jejich běh.

2.5.1 Koncepce webových služeb

Webové služby jsou postaveny na architektuře klient-server, přičemž se opírají o technologie založené na jazyce *XML* (*eXtensible Markup Language*). Výměna zpráv mezi klientem a serverem probíhá prostřednictvím protokolu *SOAP* (*Simple Object Access Protocol*). Rozhraní služeb je popsáno jazykem *WSDL* (*Web Service Description Language*). Jak již bylo zmíněno, celá síťová komunikace mezi oběma stranami probíhá protokolem *HTTP* (*Hypertext Transfer Protocol*). Webové služby mohou být navíc registrovány adresářovými službami a v nich poté vyhledávány za pomoci *UDDI* (*Universal Description, Discovery and Integration*).

Typický průběh interakce s webovou službou vypadá následovně. Poskytovatel provozuje určitou síťově dosažitelnou softwarovou komponentu – webovou službu. Vytvoří popis jejího rozhraní, které publikuje v rámci adresářové služby, či jej poskytne přímo klientské aplikaci. Klient v případě potřeby vyhledá na základě svých požadavků v registru webových služeb vyhovující webovou službu a s využitím jejího popisu s ní naváže komunikaci.

2.5.2 Srovnání webových služeb a MOM

Vzhledem k výše uvedeným skutečnostem není divu, že jsou webové služby hojně využívány jako nástroj pro zajištění interakce více aplikací a tím i tvorby distribuovaných systémů. Messaging systémy není tak snadné zakomponovat do existující architektury a navíc bývají značně nákladnější než řešení za pomoci webových služeb.[8] Webové služby však sebou nesou také jistá negativa, kvůli nimž nemusí vždy nabízet nejlepší cestu.

Webové služby ve své základní podobě postrádají dvě základní schopnosti, které mohou hovořit ve prospěch messaging systémů. Prvním nedostatkem je absence podpory transakčního zpracování, druhým je skutečnost, že webové služby negarantují doručování zpráv. Obě tyto závady se snaží řešit dodatečné specifikace webových služeb jako

jsou WS-Transaction a WS-Reliability, které však ještě nejsou na dostatečném stupni vývoje, aby mohli být široce rozšířeny, a přinášejí s sebou navíc značné zesložitění celé koncepce webových služeb.

Většina messaging systémů nabízí podporu transakcí na slušné úrovni a navíc podporuje transakce nad více datovými zdroji a aplikacemi. Spolehlivost doručování zpráv, garantující, že žádná zpráva nebude ztracena a doručena právě jednou, je navíc jedním ze základních stavebních kamenů celé koncepce messaging systémů. Vyžaduje-li tedy integrace aplikací do distribuovaného systému některou z těchto schopností, představují MOM produkty určitě vhodnější volbu.

Dalším aspektem, k němuž je nutno přihlížet při rozhodování se, jakou technologii využít, je skutečnost, že webové služby byly navrženy především pro synchronní způsob komunikace typu požadavek-odpověď. Toto omezení lze sice u webových služeb některými metodami a postupy obejít[4], avšak současná absence garance spolehlivého doručení zpráv může být pro zajištění životně důležitých procesů podniku kritická. Asynchronní komunikace a všechny výhody a nevýhody z ní plynoucí zůstávají nadále přece jen doménou systémů pro předávání zpráv.

[4]Mezi metody zavádějící do komunikace s webovými službami asynchronní prvky patří tzv. dotazování (*polling*), kdy se klient sám aktivně doptává, zda webová služba již skončila zpracovávání požadavku. Moderní implementace webových služeb (např. na platformě .NET) využívají pro asynchronní volání webových služeb tzv. metody zpětné vazby (*callback metody*), které webové služby po zpracování požadavku zavolají a které se pak postarají o zpracování vrácených výsledků.

3 Základní koncepce předávání zpráv

Jak bylo v předchozích kapitolách uvedeno, předávání zpráv (*message passing*, někdy jen *messaging*) je formou komunikace s volnou vazbou mezi softwarovými komponentami a aplikacemi, při níž dochází, jak název napovídá, k výměně zpráv mezi komunikujícími klienty.

Rozvolnění těsné vazby mezi komunikujícími aplikacemi je docíleno zavedením mezivrstvy ve formě fronty zpráv. Tento přístup způsobuje, že softwarové komponenty navzájem komunikují nepřímo, v důsledku čehož nemusí účastníci komunikace vědět o své existenci, čili odesílatelé zpráv nemusí znát jejich příjemce a ani nemusí být dostupní ve stejnou dobu. Jediné, co odesílatel a příjemce potřebují vědět, je, jaký formát zpráv mají použít, resp. očekávat a na jaké místo určení (adresu) mají zprávu zaslat, resp. z jakého místa určení si ji mají vyzvednout.

V tomto ohledu se messaging systémy značně liší od těsně svázaných technologií, jako jsou CORBA či RMI, u nichž je potřeba znát přesné rozhraní vzdálené aplikace. Je nutné také odlišovat termín předávání zpráv od elektronické pošty, jež je metodou komunikace mezi osobami nebo softwarovými aplikacemi a osobami. Předávání zpráv se používá výhradně pro komunikaci softwarových aplikací.[9]

Komunikující klienti mají rovnocenné postavení, což v praxi znamená, že libovolný klient může poslat zprávu libovolnému jinému klientu a zároveň může od libovolného klienta zprávu přijmout. Prostředky pro tvorbu, zasílání, přijímání a čtení zpráv poskytuje prostředník komunikace – messaging systém.

Messaging systémy tedy slouží pro volné svázání heterogenních systémů dohromady, přičemž současně poskytují prostředky pro zajištění spolehlivosti komunikace, transakčního zpracování a dalších vlastností. Každý systém nabízí způsoby, jak zprávy adresovat, vytvářet a jak je plnit daty. Některé systémy jsou přitom schopny rozesílat zprávy na mnoho míst určení, jiné podporují pouze posílání zpráv na jednu adresu (destinaci). Navíc některé systémy poskytují prostředky pro asynchronní příjem zpráv (tj. zprávy jsou doručeny klientovi, až jakmile dorazí), jiné podporují pouze synchronní způsob příjmu (tj. klient si musí vyžádat každou zprávu).

3.1 Případy užití předávání zpráv

Jak již bylo na mnoha místech zmíněno a naznačeno, je pole možností využití messaging systémů opravdu široké. Předávání zpráv řeší především mnohé aspekty týkající se návrhu architektury podnikových systémů. K těmto stránkám patří hlavně heterogenní integrace, škálovatelnost, zpracování událostí, souběžné zpracování požadavků a celková pružnost architektury.[10] Tato kapitola popisuje tyto nejčastější případy užití a jejich výhody.

3.1.1 Heterogenní integrace

Komunikace a integrace heterogenních platforem je snad nejklasičtějším případem použití messaging systémů. Prostřednictvím předávání zpráv lze volat služby systému z aplikací

implementovaných na kompletně odlišných platformách. Mnoho systémů pro předávání zpráv nabízí prostředky pro tvorbu klientů v různých programovacích jazycích.[5] Kromě nativního API pro jazyk, v němž je messaging systém implementován, je obvykle podporován jazyk Java prostřednictvím standardního JMS API.

Historicky bylo provozováno mnoho způsobů, jak se vypořádat s potřebou integrovat heterogenní systémy. Od přenosu informací uložených na médiích přímo od stroje ke stroji, přes využití síťových protokolů (jako např. FTP) pro přenos souborů se běžným přístupem stalo ukládání dat do sdílených databází. Vzdálené volání procedur je další cestou, jak sdílet data i funkcionalitu mezi různorodými systémy (více viz kapitola 2.3). Zatímco každé z těchto řešení má svá pro a proti, pouze předávání zpráv nabízí možnost sdílet data i funkčnost napříč aplikacemi zcela bez nutnosti vytvářet vazby a závislosti mezi nimi. Webové služby se také ukazují jako možné řešení pro propojování heterogenních systémů, nicméně zatím stále postrádají spolehlivost technologií pro předávání zpráv (viz kapitola 2.5).

Typickým příkladem je případ, kdy více systémů, které udržují svá data odděleně (např. ve vlastních databázích se zcela odlišnými schématy), potřebují udržovat jistou konzistenci mezi společnými údaji, takže dojde-li k jejich změně, je o této události spolu s novými údaji zaslána zpráva ostatním aplikacím. Ve skutečnosti je tato událost zveřejněna ve formě zprávy na určité adrese, odkud si ji může vyzvednout jakákoliv aplikace mající o ni zájem, přičemž aplikace zveřejňující zprávu vůbec nemusí vědět, jaké aplikace a kolik jich tuto zprávu obdrží. Aplikace je tak možno za provozu libovolně přidávat a odebírat, přičemž komunikačním rozhraním je vlastně messaging systém, resp. vystavené destinace.

3.1.2 Zvýšení pružnosti architektury

Použití předávání zpráv jako součásti celkové architektury podnikových systémů zlepšuje její pružnost. Tohoto je docíleno pomocí abstrakce a odstínění komponent, které s sebou předávání zpráv přináší. Pro komponenty a subsystémy tak lze zavést abstrakce až do takové míry, že mohou být nahrazeny s minimální či žádnou znalostí od klientských komponent.

Předávání zpráv umožňuje tedy vytvořit architekturu takovou, která se dokáže snadno přizpůsobit změnám, jako jsou výměna technologické platformy, výměna jednoho systému za druhý, či jejich přidání a odebrání. Prostřednictvím messaging systémů neví klientská komponenta odesílatele zprávy, v jakém programovacím jazyce, či na jaké platformě je implementována komponenta jejího příjemce, kde se tato komponenta nachází, jaký je její název a účel, či dokonce jaký protokol je použit pro přístup k ní.

3.1.3 Snížení výskytů úzkých míst systému

Úzká místa systému reprezentují slabiny, které znehodnocují především výkon celého systému. Vyskytují se kdykoliv, když jeden proces nestíhá obsluhovat požadavky na něj

[5] Toho bývá dosaženo integrací adaptérů, které konvertují zprávy, zaslané pomocí klientů různých programovacích platforem, do společného interního formátu.

činěné, což se projeví zpomalením systému v podobě prodloužení čekací doby na odpověď a může nakonec vést až k vyprchávání platnosti požadavků. Typickými reprezentanty úzkých míst systémů jsou komponenty, které mohou zpracovávat pouze limitované množství požadavků, či přistupují k omezeným zdrojům (kupř. databáze s limitovaným počtem možných souběžných spojení).

Princip, jimž se předávání zpráv vypořádává s úzkými místy systému, leží v asynchronnosti výměny zpráv. Zatímco při zpracovávání požadavků synchronní komponentou může docházet k pozdržení a hromadění požadavků, mohou být požadavky zaslané messaging systému distribuovány více komponentám, jenž se postarají o jejich zpracování. Tímto přístupem lze téměř zcela eliminovat úzká místa.

3.1.4 Zvýšení škálovatelnosti

V podobném duchu dokáží MOM produkty zvýšit celkovou škálovatelnost a propustnost systému, a tím snížit dobu odezvy na požadavky. Tohoto se dosahuje zavedením více příjemců zpráv, kteří mohou zpracovávat různé zprávy souběžně. Jinak by mohlo docházet k hromadění zpráv ve frontách, což by zvýšilo dobu odezvy a snížilo propustnost systému (fronta by se tak sama stala úzkým místem).

Další cestou ke zvýšení celkové škálovatelnosti systému je odstínění komponent systému prostřednictvím principů asynchronní komunikace. Tento přístup umožňuje růst systému do šířky, avšak hlavním limitujícím faktorem stále zůstávají systémové zdroje. Ani mnoho příjemců zpráv zpracovávajících souběžně zprávy z jediné fronty nedokáže přimět databázi k obsluze více než limitovaného množství požadavků najednou. Škálovat lze tedy pouze do jisté míry v rámci těchto praktických omezení.

3.1.5 Zvýšení produktivity uživatelů

Jak již bylo několikrát zmíněno, asynchronní způsob předávání zpráv může také napomoci ke zvýšení produktivity koncových uživatelů. Namísto aby uživatel čekal, až synchronní systém obslouží jeho (třeba i dosti náročný a zdlouhavý) požadavek, neschopen provádět jakoukoliv další činnost, použitím asynchronního předávání zpráv obdrží uživatel okamžitou odpověď indikující přijetí požadavku, načež může dále pracovat se systémem. Teprve až je požadavek zcela zpracován, je uživatel na tuto skutečnost upozorněn a jsou mu doručeny výsledky zpracování. Stráví tedy méně času čekáním, což jej činí produktivnějším. Avšak asynchronní zpracování sebou nese dodatečnou složitost.

3.1.6 Zpracování událostí

Mocným prostředkem messaging produktů je také jejich podpora možnosti odeslání jediné zprávy více příjemcům. Tento princip (označovaný jako publish/subscribe, viz 3.4.2) umožňuje, aby tutéž zprávu (a tedy tytéž data) přijalo více aplikací (i neznámého počtu) bez nutnosti, aby odesílatel explicitně rozesílal kopie zprávy jednotlivým příjemcům. O to se stará MOM systém sám.

Této vlastnosti se běžně využívá pro implementaci potřeby informovat další komponenty a subsystémy o vzniklé události. Dalším komponentám je tak dána příležitost reagovat na nastálou situaci podle vlastních potřeb. Událostí může být třeba informace o úpravě sdílených dat, upozornění na nové skutečnosti, či varování o nějaké chybě.

3.2 Architektura messaging systémů

Při asynchronním předávání zpráv používají aplikace jednoduché aplikační rozhraní pro tvorbu zpráv a jejich předání messaging systému, aby je doručil požadovaným příjemcům. Architektury MOM produktů se liší ve svých implementacích a sahají od těch centralizovaných, které závisejí na messaging serveru, který směruje zprávy, po ty decentralizované, které distribuují zpracovávání zpráv na klientské stroje. Existují dokonce i MOM produkty kombinující oba tyto přístupy.

3.2.1 Centralizovaná architektura

Messaging systémy používající centralizovanou architekturu spoléhají na messaging server, který je zodpovědný za doručování zpráv od klienta ke klientovi. Messaging server odstíňuje navzájem klienty, klienti komunikují pouze s ním, takže mohou být přidáváni a odebíráni bez dopadu na systém samotný.

Centralizovaná architektura typicky používá hvězdicovou topologii (viz obrázek 1) označovanou někdy za *hub and spoke* topologii, kde messaging server je centrálním prvkem (hub), k němuž jsou jako paprsky připojeni všichni klienti (spoke). Tato topologie umožňuje, aby každá část systému mohla komunikovat s libovolnou další částí při minimálním množství nutných síťových spojení. V reálných situacích bývá centrálním prvkem skupina (cluster) distribuovaných serverů operujících jako logická jednotka.

Tato architektura je dnes zdaleka nejrozšířenější.

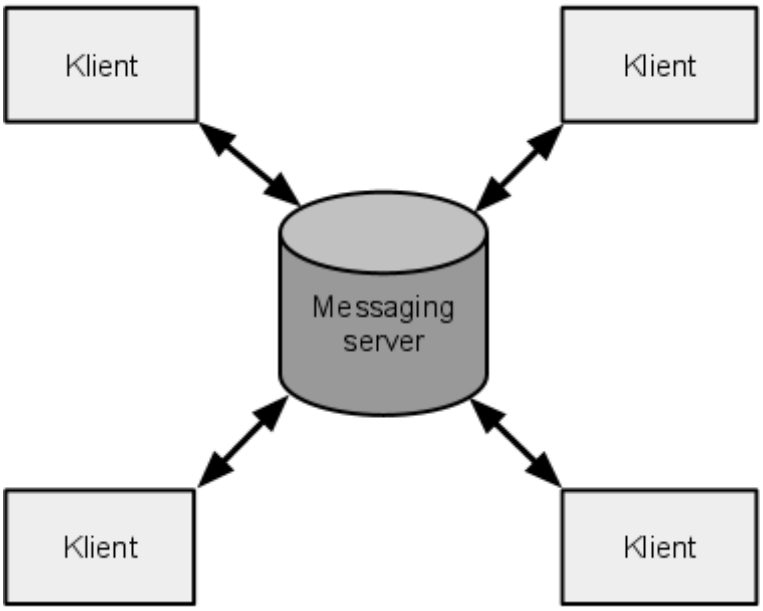
3.2.2 Decentralizovaná architektura

Messaging systémy s decentralizovanou (někdy také distribuovanou) architekturou rozesílají zprávy všem klientům v určité skupině. V takovéto architektuře nefiguruje žádný centrální messaging server. Jeho funkcionalita, jako jsou perzistence, transakce a bezpečnost, je částečně vložena jako součást klienta, zatímco směrování zpráv je ponecháno na síťové vrstvě použitím protokolu pro IP multicast (viz obrázek 2).

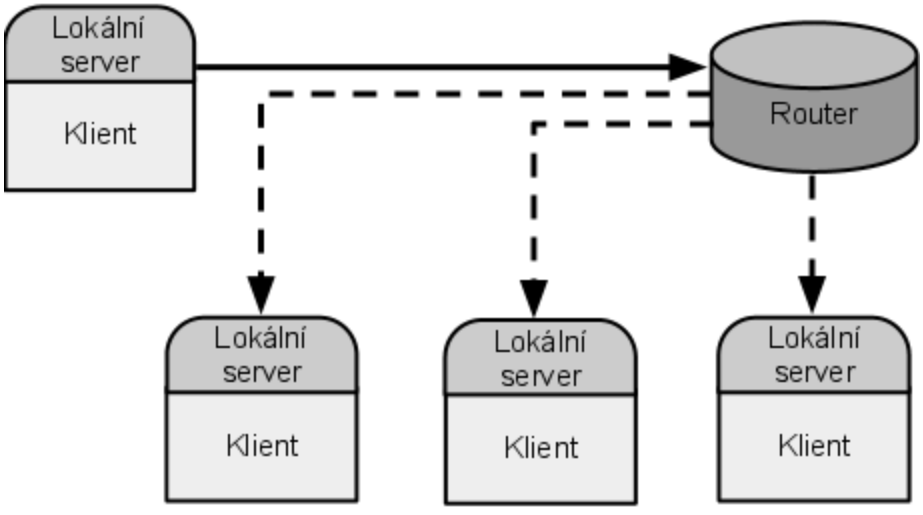
IP multicast umožňuje aplikacím připojit se do jedné či více skupin, kde každá skupina má svou síťovou adresu, takže všechny zprávy zaslané na určitou adresu budou doručeny všem členům příslušné skupiny. Centrální server není tedy pro účely směrování zpráv potřeba, o to se stará síť automaticky.

3.3 Návrh messaging systémů

Odpovídají-li požadavky na systém některému případu užití messaging systému, ovlivní rozhodnutí o použití některého MOM produktu návrh celého systému. Messaging systém



Obrázek 1: Centralizovaná architektura s hvězdicovou topologií



Obrázek 2: Decentralizovaná architektura využívající multicast

je umístěn mezi dvěma komunikujícími komponentami jako prostředník, který zajišťuje předávání zpráv mezi těmito komponentami, čímž potlačuje přímou vazbu mezi nimi, takže jsou tyto komponenty od sebe navzájem odstíněny. Komponenty tak tvoří spolu s messaging systémem dříve zmiňovanou hvězdicovou topologii.

Dalším aspektem, s nímž je třeba se při návrhu systémů využívajících předávání zpráv vypořádat, je asynchronnost komunikace. Ta celkově komplikuje návrh, neboť obecně způsobuje, že požadované výsledky nejsou ihned dostupné, ale až za nějakou dobu a na jiném místě aplikace. Odpovědi bývají totiž většinou doručovány zvláštním, k tomuto účelu určeným komponentám, běžícím odděleně (a mnohdy i zcela nezávisle) od komponent, které zprávy požadavků odeslali. Toto vede k rozbití toku zpracování a možné ztrátě kontextu. Aplikační rozhraní většiny messaging produktů obsahuje prostředky pro tvorbu komponent pro asynchronní příjem zpráv. Podle JMS specifikace jsou tyto komponenty označovány jako posluchači zpráv (více viz kapitolu 4.7.5).

Fronty a témata zpráv tak tvoří jakési komunikační rozhraní mezi aplikacemi. Fungují jako takové schránky, kam jedna aplikace uloží zprávu ve specifickém formátu a odkud si ji jiná vyzvedne. Na této abstrakci staví několik principů pro řešení integrace podnikových aplikací (*Enterprise Application Integration*, *EAI*), populární architektonický styl známý jako na služby orientovaná architektura (*Service Oriented Architecture*, *SOA*), či již dříve představená ESB.

3.4 Modely komunikace

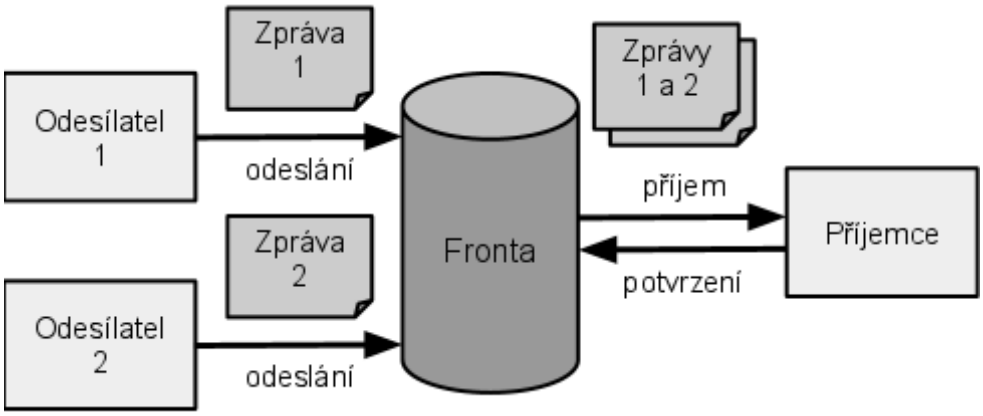
V oblasti messaging systémů se běžně používají dva modely komunikace označované jako messaging domény. Jsou jimi *point-to-point* model a model *publish/subscribe*. Messaging domény se liší nejen počtem klientů odesílajících a přijímajících zprávy, pro než se zavádí označení producenti (*message producer*), resp. konzumenti (*message consumer*) zpráv, ale také způsobem, jakým si konzumenti své zprávy vyzvedávají a jak jsou zprávy produkovány.

3.4.1 Point-to-point model

Point-to-point (PTP) messaging doména je postavena na konceptu fronty zpráv (*queue*). Odesílatel adresuje každou zprávu na jistou frontu a příjemce si je z této fronty vyzvedává. V tomto modelu odesílatel zná místo určení zprávy, a zasílá tedy zprávu přímo do fronty příjemci. Fronta přitom uchovává všechny přijaté zprávy, dokud nejsou vybrány příjemcem anebo dokud nevyprší jejich platnost. Situace je znázorněna na obrázku 3.

V doméně PTP má každá zpráva pouze jednoho příjemce. Jakmile je přijatá zpráva úspěšně zpracována, příjemce odešle její potvrzení. Dojde-li k systémovému selhání, dříve než messaging systém obdrží potvrzení příjetí zprávy, tak se messaging systém po obnovení systému pokusí o opětovné doručení zprávy. Proběhne-li však proces potvrzování zprávy úspěšně, odstraní messaging systém zprávu z fronty, a nelze ji tedy už znovu doručit.

Mezi odesílatelem a příjemcem neexistuje žádná časová závislost, a proto nemusí odesílatel a příjemce zprávy běžet současně (a nemusí ani vědět o své vzájemné existenci),



Obrázek 3: Point-to-point model

což znamená, že příjemce může přijmout zprávu bez ohledu na to, zda odesílatel zprávy v danou chvíli běží či nikoliv.

Příkladem modelu PTP může být objednávkový systém. Zákazníci vytvářejí své objednávky na zboží, které jsou postupně řazeny do fronty nevyřízených objednávek. Z této fronty si je pak příslušná zodpovědná osoba vyzvedává a postupně je vyřizuje. Nevyřízené objednávky může z fronty vybírat a obsluhovat více osob, ale nikdy žádné dvě osoby nevyřizují tutéž objednávku.

3.4.2 Model publish/subscribe

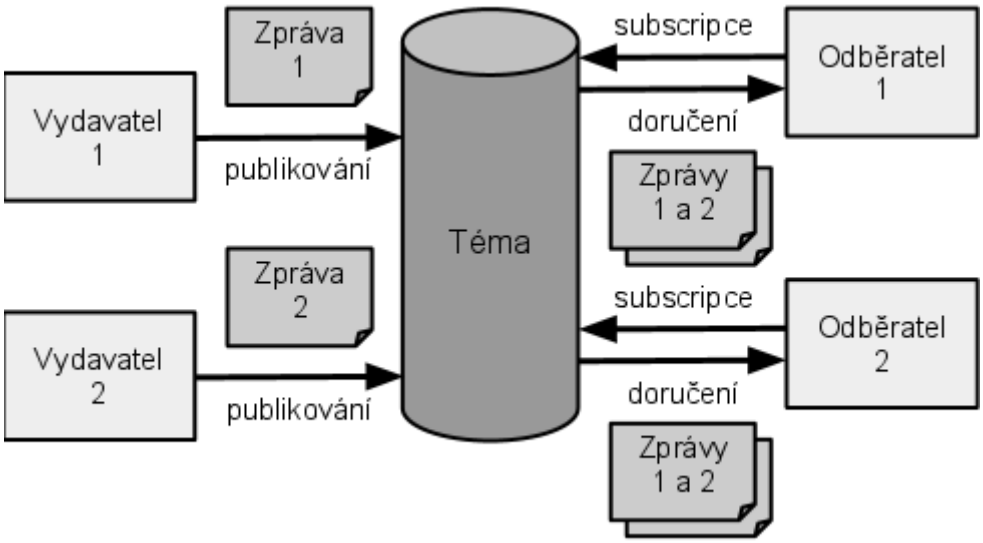
V publish/subscribe (Pub/Sub) doméně klienti adresují zprávy na určitý uzel v kontextové hierarchii, pro nějž bylo zavedeno označení téma zpráv (*topic*). V této doméně se producentům zpráv říká vydavatelé (*publisher*) a konzumentům odběratelé (*subscriber*).

Odběratelé vyjadřují svůj zájem o odběr zpráv z určitého tématu přihlášením se k odběru (neboli vytvořením subskripce – *subscription*) na tomto tématu. Všichni registrovaní odběratelé poté obdrží kopii každé zprávy publikované vydavatelem k tématu. Jediná zpráva je tedy přijata více klienty. Tuto situaci zachycuje obrázek 4.

V Pub/Sub doméně tedy smí mít každá zpráva více odběratelů. Odběratelé a vydavatelé bývají anonymní a jejich počet u tématu se může v čase měnit. Opět tedy mezi nimi neexistuje časová závislost.

Messaging systém distribuuje zprávy přicházejících k tématu pouze ke všem současně registrovaným odběratelům. Klient, který se přihlásí k odběru zpráv z určitého tématu, může zkonzumovat pouze ty zprávy, které byly publikovány až po vytvoření subskripce, přičemž musí zůstat nepřetržitě aktivní.

Některé messaging systémy umožňují jisté uvolnění této podmínky zavedením tzv. trvanlivých subskripcí. Trvalé subskripce (*durable subscriptions*) dovolují registrovaným odběratelům přijímat zprávy i v době, kdy nejsou aktivní. Trvanlivé subskripce poskytují flexibilitu a spolehlivost ve stejné míře, v jaké je nabízejí fronty zpráv modelu PTP (avšak



Obrázek 4: Publish/subscribe model

s možností rozesílat zprávy více příjemcům). Trvanlivé subskripce totiž uchovávají kopii každé zprávy publikované k tématu, dokud se odběratelé nestanou opět aktivními a nemohou si je vyzvednout. Dokonce i v případě systémového selhání.

Jako příklad modelu Pub/Sub může posloužit publikační systém novinek. Různí vydavatelé zveřejňují své články k různým tématům, zatímco se čtenáři registrují u témat, o něž mají zájem, k odběru zveřejněných článků. Informace o nově zveřejněném článku určitého tématu je tak zaslána všem čtenářům přihlášeným k jeho odběru.

3.5 Messaging protokoly a aplikační rozhraní

Vzhledem k neexistenci jednotného standardu komunikace klientských aplikací s messaging systémy se k tomuto účelu používá celá řada prostředků. Velké množství messaging systémů poskytuje vlastní proprietární aplikační rozhraní a komunikační protokol. Tento přístup většinou znemožňuje spolupráci různých messaging systémů, stejně jako přenositelnost klientských aplikací či snadnou změnu MOM produktů. Naštěstí existuje v této oblasti softwaru několik rozšířených standardů a standardních způsobů interakce s messaging systémy.

3.5.1 Proprietární API

Jak již bylo uvedeno, mnoho MOM produktů poskytuje vlastní programové prostředky pro interakci s messaging systémem. Výhodou těchto nástrojů je, že aplikačním klientům plně zpřístupňují veškerou funkcionalitu systému, takže klienti mohou využít všechny prostředky a ovlivnit všechny aspekty, které messaging systém nabízí.

Obecná rozhraní, jakým je třeba JMS, nejsou obvykle dostatečně bohatá, aby jimi bylo možno využít všechny speciální vlastnosti, které mnohé messaging systémy poskytují.

3.5.2 JMS API

JMS je součástí specifikace platformy Java EE firmy Sun. Jedná se o specifikaci sady rozhraní programovacího jazyka Java s přesně definovanou sémantikou (významem, účelem a vzájemnými vztahy), která určuje způsoby, jakými klienti přistupují k prostředkům messaging systémů a využívají jejich služeb. JMS bylo vytvořeno jako specifikace nejmenšího společného jmenovatele zapouzdřující společnou funkcionalitu existujících podnikových systémů pro zasílání zpráv, jenž byli dostupní v době jejího vzniku roku 2002.

JMS ve specifikaci nedefinuje komunikační protokol (tj. formát přenášených dat komunikace), pouze definuje programové rozhraní, takže klienti messaging systémů od různých výrobců nemohou přímo spolupracovat, protože každý používá odlišný, vlastní přenosový protokol.

JMS se stalo velice populárním API a je implementováno většinou MOM produktů. Stalo se tak de facto standardem pro interakci s těmito systémy. JMS je však dostupné pouze klientům běžícím na platformě Java. Principům JMS je věnována celá kapitola 4.

3.5.3 NMS API

NMS (*.NET Message Service*) API poskytuje standardní rozhraní pro práci s messaging systémy na platformě .NET. Umožňuje tak vytvářet messaging aplikace prostřednictvím programovacích jazyků této platformy (C#, Visual Basic a další) a zároveň využívat bohatství jejích knihoven.

NMS API bylo vyvinuto firmou Apache Software Foundation, přičemž se tvůrci inspirovali specifikací JMS tak, že v současné době podporuje všechny jeho prvky (včetně transakcí, trvanlivých subskripcí atd.). Aktuálně je však implementována podpora pouze u několika messaging produktů – Apache ActiveMQ, Microsoft Message Queue (MSMQ), či TIBCO EMS. Rovněž je zahrnuta podpora protokolu STOMP.[11]

3.5.4 RESTful API

Representional State Transfer (*REST*) je architektura rozhraní, navržená pro distribuované prostředí. Rozhraní REST je použitelné pro jednotný a snadný přístup ke zdrojům (data či stavy aplikace). Zdrojem jsou identifikovány svým URI a REST definuje způsoby přístupu k nim prostřednictvím tzv. *CRUD* metod.[6] Tyto metody jsou většinou implementovány pomocí operací HTTP protokolu.[7]

REST díky své jednoduchosti získal v nedávné době na oblibě a jeho přístup k zasílání zpráv se jeví jako vážný kandidát na získání statutu standardu pro zajištění spolupráce

[6] *Create*, *Read*, *Update* a *Delete* (zkráceně *CRUD*) je označení pro čtyři základní operace pro práci s persistentními úložišti a daty vůbec. Reprezentují operace pro vytvoření, čtení, úpravu a odstranění záznamu.

[7] Jednotlivým CRUD operacím pak odpovídají v HTTP protokolu metody GET, POST, PUT a DELETE.

mezi systémy. Používá se například i pro implementaci webových služeb (tzv. RESTful Web Services). Aplikační rozhraní založené na principech architektury REST se stává součástí stále více MOM produktů, poskytující jim tak jednoduché, změnám odolné rozhraní, jenž se dá snadno rozšířit.[12]

3.5.5 AMQP

Advanced Message Queuing Protocol (*AMQP*) je otevřeným standardním protokolem pro MOM. AMQP definuje formát dat přenášených při komunikaci mezi klientem a messaging systémem, takže implementace od jiných výrobců dokáží spolupracovat.

Každý nástroj (klient, komponenta), který dokáže vytvořit a interpretovat zprávy odpovídající formátu AMQP, dokáže komunikovat a spolupracovat s jakýmkoliv dalším vyhovujícím nástrojem bez ohledu na programovací jazyk implementace. Tímto se otevírá cesta k tvorbě distribuovaných systémů nezávislých na platformě a implementaci.

AMQP získává rychle na popularitě, a je proto implementován mnohými MOM produkty. Zároveň byla vytvořena řada klientů pro interakci s AMQP poskytovateli v mnoha různých programovacích jazycích.[13]

3.5.6 STOMP

STOMP (*Streaming Text Orientated Messaging Protocol*), dříve známý jako TTMP, je jednoduchý textový protokol navržený pro práci se systémy pro zasílání zpráv. Definuje přenosový protokol, který umožňuje jakémukoliv STOMP klientovi komunikovat s libovolným messaging systémem, který STOMP podporuje (tzv. *STOMP Message Broker*), bez ohledu na jazyk a platformu, v nichž jsou klienti a STOMP poskytovatel implementováni.

Pro STOMP byl vytvořen adaptér StompConnect, který zabaluje STOMP komunikaci do zpráv podle JMS specifikace, takže lze STOMP klienty využít pro interakci s libovolným JMS poskytovatelem. Spolu s faktem, že vzhledem k jednoduchosti STOMP protokolu bylo implementováno množství klientů pro různé platformy, tak umožňuje rozšířit působnost JMS messaging systémů i mimo platformu Java.

Protokol STOMP byl implementován řadou messaging produktů a s pomocí adaptéru StompConnect jej lze využít se všemi JMS poskytovateli, a tedy téměř se všemi většími messaging systémy vůbec. K systémům s nativní podporou protokolu STOMP patří Apache ActiveMQ, HornetQ, RabbitMQ, MorbidQ, Gozirra či StompServer.[14]

Principy STOMP protokolu

Protokol STOMP je podobný protokolu HTTP (a také vystupuje na aplikační vrstvě TCP/IP síťového modelu) a pracuje nad některým protokolem (obvykle TCP) transportní vrstvy. Navíc obsahuje podporu transakčního zpracování rámců.

Komunikace mezi klientem a serverem probíhá prostřednictvím tzv. rámců, jenž se skládají z několika řádků. První řádek obsahuje vlastní příkaz, dále jsou uvedeny hlavičky zprávy ve formátu `<klíč>: <hodnota>`. Hlavička je od těla s vlastním obsahem zprávy oddělena prázdným řádkem. Tělo a tím i celý rámec je ukončeno znakem NULL (Unicode

hodnota \u0000).[15] Mnohé rámce mívají prázdné tělo. Jedná se o rámce administračních příkazů, u nichž jsou důležité informace obsaženy v hlavičkách (např. `CONNECT`, `SUBSCRIBE`, `BEGIN` atd.).

Průběh komunikace

Tabulky 1 a 2 představují typický průběh komunikace mezi klientem v roli producenta, resp. konzumenta zpráv se STOMP messaging serverem.[8]

	Klient	**Server**

1	`CONNECT`
	`login: hal191`
	`passcode: 191lah`
	
	`^@`
		Navázání spojení klientem. Za účelem autentizace klienta je předáváno uživatelské jméno (`login`) a heslo (`passcode`).

2	Potvrzení spojení serverem. Přiřadí této relaci jednoznačný identifikátor (`session`).	`CONNECTED`
		`session: session-12345`
		
		`^@`

3	`SEND`
	`destination: fronta`
	
	`Ahoj, světe!`
	`^@`
		Odeslání zprávy „Ahoj, světe!“ na místo určení (`destination`) s názvem „fronta“.

4	`DISCONNECT`
	
	`^@`
		Uzavření spojení na server.

Tabulka 1: Odeslání zprávy protokolem STOMP

Kroky navazování spojení (tj. 1 a 2) jsou při komunikaci mezi konzumentem zpráv a serverem shodné. Tentýž je vždy i poslední krok komunikace (4, resp. 6) – uzavření spojení.

	Klient	**Server**

3	`SUBSCRIBE`
	`destination: fronta`
	`ack: auto`
	
	`^@`
		Registrování klienta jako odběratele zpráv z místa určení (`destination`) s názvem „fronta“. Zprávy budou potvrzovány (`ack`) automaticky při přijetí.

[8] Symboly `^@` reprezentují v příkladech znak NULL.

	Klient	**Server**

4	Zaslání zprávy s identifikátorem (`messagi-id`) „message-12345“ z místa určení (`destination`) „fronta“ a obsahem „Ahoj, světe!“.	`MESSAGE`
		`destination: fronta`
		`message-id: message-12345`
		
		`Ahoj, světe!`
		`^@`

5	`UNSUBSCRIBE`
	`destination: fronta`
	
	`^@`
		Odhlášení klienta jako odběratele zpráv z místa určení (`destination`) s názvem „fronta“.

Tabulka 2: Příjem zprávy protokolem STOMP

4 Java Message Service

Java Message Service je specifikací aplikačního rozhraní (API) jazyka a platformy Java, navržená firmou Sun Microsystems, která umožňuje aplikacím napsaných v jazyce Java komunikovat s middleware produkty pro zasílání zpráv. JMS specifikace tedy definuje pro platformu Java, Enterprise Edition jednotný způsob, jak vytvářet, posílat, přijímat a číst zprávy podnikových messaging systémů, přičemž nabízí podporu pro obě komunikační domény, tedy pro komunikaci typu point-to-point i publish/subscribe (včetně trvanlivých subskripcí).

Sada rozhraní JMS API byla navržena tak, aby postihovala společné koncepce a prvky podnikových messaging systémů existujících v době jejího vzniku. Toto vedlo ke snadnému přijetí JMS specifikace u těchto společností a její rychlé rozšíření a implementaci do mnoha messaging produktů.

Přestože je JMS API ve své snaze postihnout společné a nikoliv všechny prvky messaging systémů vcelku jednoduché, tak stále nabízí dostatek prostředků pro tvorbu robustních a sofistikovaných messaging aplikací. Tento přístup minimalizuje nároky kladené na znalosti vývojářů, aby mohli využívat i pokročilé prvky messaging systémů, a zároveň tak maximalizuje přenositelnost vytvořených JMS aplikací napříč messaging systémy podporujícími JMS (tzv. poskytovateli JMS).

JMS API nabízí prostředky pro zajištění asynchronní a spolehlivé komunikace mezi volně svázanými aplikacemi. Asynchronní znamená, že si klient nemusí vyžádat přijetí každé zprávy, ale je mu doručena, jakmile dorazí. Za spolehlivou je považována taková komunikace, kdy je každá zpráva doručena právě a jen jednou, není tedy ztracena, ani doručena vícekrát. Pro aplikace, které si mohou dovolit přijít o některé zprávy či přijmout zprávy duplicitně, nabízí JMS API nižší úrovně spolehlivosti.

4.1 Historie vývoje JMS specifikace

JMS specifikace byla vyvinuta společností Sun Microsystems spolu s několika dalšími partnerskými společnostmi (především tvůrců MOM produktů) pod záštitou Java Community Process (JCP) jako specifikace JSR (Java Specification Request) 914.

JMS specifikace byla poprvé představena v srpnu roku 1998, přičemž JMS API bylo ve verzi 1.0 zařazeno do platformy J2EE 1.2. První stabilní referenční implementace JSM API (verze 1.0.2b) byla dostupná v červnu roku 2001, přičemž se stala součástí platformy J2EE 1.3. Zatím poslední verzí specifikace JMS je verze 1.1 vydaná v květnu 2002, která zavedla mnohé změny, přičemž zachovala zpětnou kompatibilitu s předchozí verzí, a jejíž referenční implementace je nabízena na platformě J2EE (resp. Java EE) od verze 1.4.[16]

4.2 Funkcionalita nepodporovaná JMS

V této sekci je výčet vlastností, které specifikace JMS neobsahuje či nedefinuje, a nelze je tak prostřednictvím JMS API využít. To však neznamená, že tyto schopnosti ani nejsou podporovány messaging systémy implementujícími JMS specifikaci. Pro jejich využití je

však nutno aplikovat jiné prostředky, většinou nástroje dodávané spolu se samotným messaging systémem.

Jak už by mělo být zřejmé z předcházejících kapitol, JMS API není rozhraním pro elektronickou poštu, přestože funguje na podobném principu. Základem komunikace mezi klienty v rámci JMS jsou asynchronně posílané zprávy (požadavky, události, odpovědi apod.). Tyto zprávy jsou vytvářeny a užívány aplikacemi a obsahují informace, díky kterým tyto aplikace fungují a koordinují svou činnost. Zprávy elektronické pošty jsou sice také posílány aplikacemi, ale jejich primárním cílem jsou uživatelé (tj. lidé).[1]

Vyrovnávání zátěže (Load Balancing) a tolerance chyb (Fault Tolerance) – JMS API nedefinuje způsoby kooperace mnoha klientů implementujících nějakou kritickou službu systému.

Chybová a informativní oznámení (Error and Advisory Notification) – JMS nestandardizuje zprávy, jimiž mnohé messaging systémy asynchronně oznamují klientům výskyt chyb či systémových událostí. JMS specifikace ukazuje cestu, jak se lze vyhnout použití těchto zpráv, a tedy jak zachovat přenositelnost klientů.

Administrace – JMS API nedefinuje způsob správy a řízení messaging systémů, zajišťuje pouze komunikaci.

Bezpečnost – JMS API nedefinuje rozhraní pro zajištění zabezpečení a integrity zpráv. Bezpečnost je ponechána plně v kompetenci JMS poskytovatelů, takže je tedy konfigurována administrátory spíše než klienty. Způsobem zabezpečování messaging systémů se zabývá kapitola 5.

Komunikační protokol (Wire Protocol) – JMS nedefinuje protokol komunikace (tj. přenášená data a jejich formát), takže klienti různých JMS poskytovatelů, kteří využívají různý (většinou vlastní nestandardní) protokol přenosu dat, nedokáží spolupracovat. JMS specifikace však stanovuje, že by si JMS poskytovatelé měli být schopni poradit (ačkoliv třeba ne tak efektivně) i se zprávami s cizí implementací. Tento pokyn však bývá JMS poskytovateli dosti často nerealizován.

Úložiště typů zpráv (Message Type Repository) – JMS nedefinuje způsob uchovávání definic typů zpráv, ani jazyk pro tvorbu těchto definic.

4.3 Součinnost JMS API a technologií platformy Java EE

Když bylo v roce 1998 JMS API představeno, jeho důležitým účelem bylo umožnit Java aplikacím přistupovat k existujícím MOM produktům. Od zveřejnění specifikace platformy J2EE verze 1.2, byli J2EE poskytovatelé (tj. implementace poskytovatelů služeb J2EE platformy) povinni poskytovat JMS API rozhraní, ale nikoliv implementovat jej. Tento požadavek přišel až se specifikací platformy J2EE ve verzi 1.4.[1]

Zařazení JMS API do specifikace Java EE platformy ji obohacuje o možnost vytvářet volně svázané, spolehlivé a asynchronní interakce mezi Java EE komponentami a messaging systémy. Aplikační vývojáři mohou tedy bez rozpaků využít při tvorbě komponent

pro zasílání zpráv i jiná aplikační rozhraní obsažená ve specifikaci platformy Java EE. Tato kapitola zmiňuje součinnost JMS API s těmi nejčastěji používanými Java EE technologiemi.

4.3.1 Java DataBase Connectivity

JDBC představuje jednotné aplikační rozhraní pro přístup k relačním databázím. Interakce s relačními databázemi je častým případem užití nejen JMS klientů. Ti navíc mohou požadovat použití JMS API a JDBC API v jediné transakci. Toho lze dosáhnout automaticky implementováním klientů jako komponent *Enterprise JavaBeans*, či přímo prostřednictvím *Java Transaction API* (*JTA*).

4.3.2 Komponenty JavaBeans

JavaBeans představují jednoduché softwarové komponenty (familiárně označované jako beany) podléhajícím určitým pravidlům tvorby používané většinou pro tvorbu prezentační vrstvy webových aplikací. Komponenty JavaBeans mohou sice využít JMS pro odesílání a příjem zpráv, ale samo JMS API a rozhraní, která definuje, nebylo navrženo pro tento způsob použití.

4.3.3 Komponenty Enterprise JavaBeans

Enterprise JavaBeans (*EJB*) jsou řízené, serverové komponenty pro tvorbu modulárních distribuovaných podnikových aplikací. Tyto komponenty se většinou používají k implementaci aplikační a perzistentní vrstvy podnikových systémů. Současná verze specifikace EJB 3.1 definuje dva základní typy EJB komponent. Relační beany *Session Beans* umožňují synchronní komunikaci na způsob RPC, zatímco zprávami řízené beany *Message Driven Beans* (*MDB*) slouží k asynchronní komunikaci.

EJB komponenty ve spojení s JMS API a dalšími zdroji jako JDBC tvoří silnou kombinaci pro tvorbu podnikových služeb. Oba typy komponent mohou prostřednictvím JMS API zprávy odesílat a MDB komponenty mohou zprávy navíc přijímat. Přímo k tomuto účelu byly primárně navrženy. Více o MDB EJB komponentách viz kapitola 4.11.2.

4.3.4 Java Transaction API

Balíček javax.transaction, označovaný jako *Java Transaction API* (*JTA*) poskytuje klientské aplikační rozhraní pro ohraničení distribuovaných transakcí a umožňující zdrojům účastnit se distribuovaných transakcí.

Přistupují-li JMS klienti k dalším zdrojům, mohou využít JTA právě pro demarkaci distribuovaných transakcí. Nicméně se nejedná o funkci samotného JMS, ale o funkci transakčního prostředí, v němž klient běží.

4.3.5 Java Transaction Service

Java Transaction Service (*JTS*) je specifikace pro implementaci transakčních managerů, kteří slouží k řízení a monitorování distribuovaných transakcí mezi aplikacemi a zdroji podporujícími transakce jako jsou databázové servery a messaging systémy. JTS specifikace zahrnuje i specifikaci JTA.

JMS může využít JTS k explicitní specifikaci distribuovaných transakcí, avšak vhodnější je využít automatických prostředků poskytovaných JMS klientů jako EJB komponent.

4.3.6 Java Naming and Directory Interface

Java Naming and Directory Interface (*JNDI*) je API poskytující jednotný přístup k adresářovým a jmenným službám jako jsou např. LDAP adresáře. Tyto služby se používají jako úložiště objektů, k nimž se přistupuje prostřednictvím přiřazeného jména.

V JMS aplikacích se JNDI využívá jako standardní způsob pro získávání administrovaných objektů, tj. objektů vytvářených a konfigurovaných administrátorem pomocí nástrojů dodávaných s messaging systémy. Delegování činností závislých na poskytovateli na administrátora zvyšuje přenositelnost a spravovatelnost vytvořených JMS klientů.

4.4 JMS poskytovatelé

Jak již bylo zmíněno, všechny Java EE aplikační servery od verze 1.4 musí obsahovat implementaci JMS poskytovatele. Tato kapitola obsahuje stručný a neúplný seznam JMS poskytovatelů, tedy produktů implementujících JMS specifikaci a umožňujících tak přístup k prostředkům messaging systémů prostřednictvím JMS API. K dispozici je celá řada volně dostupných (open-source) i komerčních messaging systémů implementujících JMS specifikaci a poskytujících různé sady vlastností, výkonnostní možnosti a úrovně kvality dokumentace, nástrojů a podpory.[17]

4.4.1 Open-source produkty

Následuje přehled JMS messaging systémů distribuovaných pod různými typy veřejných licencí.

- Apache ActiveMQ – implementace od Apache Software Foundation nabízející některé vyspělejší prvky (clusterování, různé způsoby perzistence atd.) a řadu klientů pro různé programovací jazyky (C#, C/C++, Delphi, PHP, Perl, Python, Ruby a další)
- FUSE Message Broker – poskytovatel založený na Apache ActiveMQ od firmy Progress Software pro tvorbu rozsáhlé podnikové infrastruktury pro propojení heterogenních systémů
- OpenJMS – implementace JMS nezávislá na použitém aplikačním serveru, a proto může posloužit jako společné rozhraní pro klienty různých výrobců

- JBoss Messaging – modulární messaging systém, který slouží jako implicitní JMS poskytovatel na aplikačních serverech JBoss do verze 5
- HornetQ – implementace od divize JBoss společnosti Red Hat, Inc. původně označovaná jako JBoss Messaging 2.0, která však nabízí podporu více protokolů, vysoký výkon, clusterování, možnost napojení na jiné JMS servery a další vlastnosti
- JORAM – produkt konsorcia OW2 implementující protokol AMQP a poskytující některé pokročilé prvky včetně Java ME či C/C++ klienta
- OpenMQ (Open Message Queue) – poskytovatel od tvůrců JMS specifikace firmy Sun nabízející dodatečné prvky pro podniková řešení a integrovaný jako implicitní JMS poskytovatel do aplikačního serveru GlassFish
- RabbitMQ – messaging systém divize SpringSource společnosti VMWare, Inc. napsaný v jazyce Erlang a postavený na platformě OTP (Open Telecom Platform), používající protokol AMQP

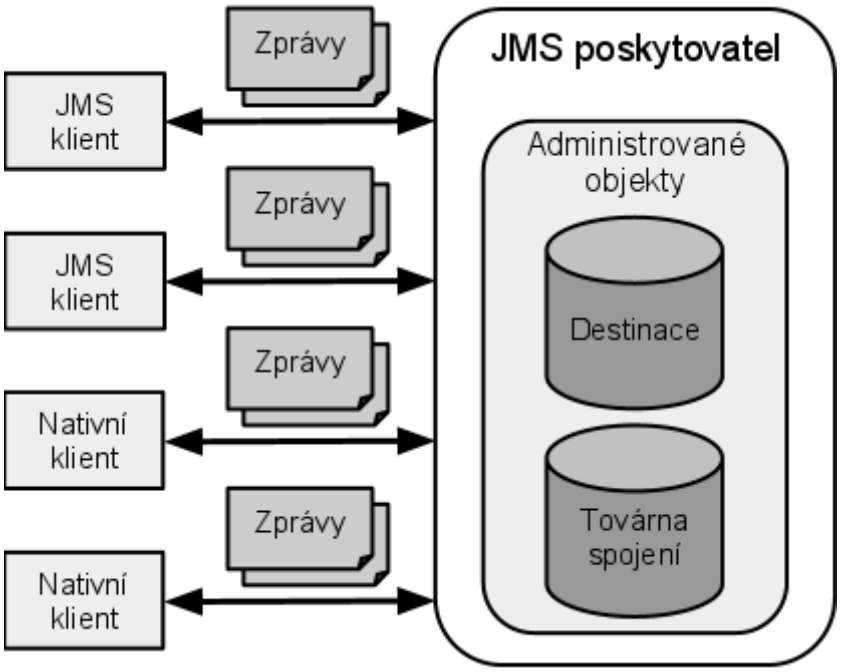
4.4.2 Komerční produkty

Komerční JMS poskytovatelé obvykle nabízejí komplexní řešení (většinou v podobě balíků zahrnujících aplikační server, databáze a další produkty) pro zajištění spolupráce podnikových systémů spojených s vyšší úrovní uživatelské podpory.

- BEA Weblogic – produkt původně vytvořen firmou BEA System, Inc., nyní součást balíku Oracle Fusion Middleware společnosti Oracle Corporation
- Oracle AQ (Oracle Advanced Queuing) – MOM produkt vyvinut společností Oracle Corporation integrovaný do databáze Oracle
- SAP NetWeaver WebAS – aplikační server korporace SAP AG implementující JMS
- SonicMQ – vyspělý MOM produkt od společnosti Progress Software
- TIBCO Rendezvous – produkt pro integraci podnikových aplikací od společnosti TIBCO Software
- WebSphere MQ – podnikový messaging systém, jenž je součástí softwarové rodiny produktů společnosti IBM

4.5 Architektura JMS aplikace

JMS je postaveno na několika základních stavebních blocích, které vystupují v každé JMS aplikaci. Tyto základní elementy architektury JMS aplikací a jejich vztah zachycuje obrázek 5.



Obrázek 5: Architektura JMS aplikace

JMS poskytovatel – softwarová entita vytvořená podle JMS specifikace pro potřeby MOM produktu, aby umožnila jeho spolupráci s aplikacemi vytvořených prostřednictvím JMS API. Jedná se tedy o vlastní messaging systém implementující JMS rozhraní spolu s dalšími nástroji (JMS specifikací explicitně nedefinovaných) poskytujícími administrativní a řídící funkce od plnohodnotného messaging produktu vyžadované. Poskytovatelé jsou implementováni čistě v jazyce Java, nebo hrají úlohu adaptéru k messaging produktům mimo platformu Java.

JMS klienti – aplikace či komponenty napsané v programovacím jazyce Java, které posílají nebo přijímají zprávy.[9]

Nativní klienti – klienti, kteří používají místo JMS API nativní klientské aplikační rozhraní daného messaging systému. Může se také jednat i o klienty vytvořené v jiném programovacím jazyce než Java.[10]

JMS zprávy – objekty přenášené mezi JMS klienty, které obsahují komunikační data. Představují tedy vlastní předmět komunikace klientů.

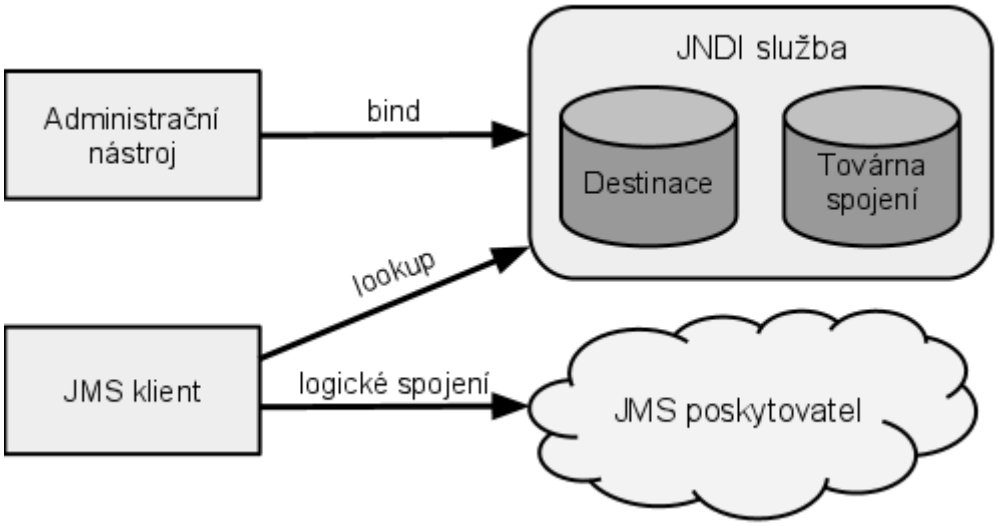
[9]Podle způsobu použití JMS API rozlišujeme dva typy klientů. Producentem JMS (*JMS Producer*) je JMS klient, který vytváří a odesílá JMS zprávy. Konzumentem JMS (*JMS Consumer*) je typ JMS klienta, který JMS zprávy přijímá.

[10]Pokud byla aplikace využívající služeb messaging systému vytvořena před zavedením podpory JMS API, je pravděpodobné, že bude používat oba typy klientských aplikací, tedy JMS i nativní (ne-Java) klienty.

Administrované objekty – přednastavené JMS objekty vytvořené administrátorem, určené pro použití klienty. Dvěma typy administrovaných objektů potřebných v JMS aplikacích jsou továrny spojení a místa určení (neboli destinace – *destination*), které jsou blíže popsány v kapitole 4.7.

Jednotliví JMS poskytovatelé se značně liší nejen v technologii použité pro zasílání zpráv, ale také ve způsobu instalace a administrace. Jestliže JMS klienti mají být přenositelní, musí být od těchto proprietárních aspektů messaging systémů odstíněni. Této izolace je dosaženo definováním JMS administrovaných objektů, které jsou vytvářeny a nastavovány pomocí administračních nástrojů poskytovatele. Klienti je pak používají prostřednictvím přenositelných rozhraní JMS API.[1]

Administrované objekty jsou obvykle administrátorem umístěny do JNDI jmenného prostoru (operace bind), kde poté mohou být vyhledány JMS klienty (lookup) a použity pro vytvoření logického spojení na tyto objekty prostřednictvím JMS poskytovatele. Obrázek 6 ilustruje tuto interakci.



Obrázek 6: Technika práce s administrovanými objekty

4.6 Konzumace zpráv

Jednou z důležitých koncepcí messaging systémů je způsob konzumace zpráv. Konzumace zprávy je proces zahrnující příjem zprávy a přečtení zprávy neboli vyzvednutí zprávy z místa určení (fronty nebo tématu). Producentem zprávy (odesílatelem či vydavatelem) se nestará o způsob její konzumace konzumentem. Tento proces je v kompetenci přijímající strany, která je cílem zprávy.

Ačkoliv produkty pro zasílání zpráv jsou neodmyslitelně asynchronní, takže mezi produkcí a konzumací zprávy neexistuje žádná časová závislost, tak JMS specifikace

definuje, že zprávy mohou být zkonzumovány nejen asynchronním způsobem, ale i synchronně.

Synchronní konzumace zprávy – Při vyzvedávání zprávy z místa určení čeká příjemce zprávy, dokud zpráva nedorazí, což v praxi znamená, že až do příchodu zprávy blokuje chod aplikace, která v průběhu této blokace neprovádí žádné jiné činnosti. Tento způsob konzumace zpráv je vhodný v případech, kdy je přijetí zprávy nutnou podmínkou pro následný chod aplikace.

Kvůli prevenci možného uváznutí v případě neschopnosti doručit očekávanou zprávu umožňuje JMS API specifikovat časový limit, po jehož vypršení dojde k opětovnému odblokování aplikace.

Asynchronní konzumace zprávy – Na návrhovém vzoru pozorovatel (*observer*) je založen princip, jehož prostřednictvím může klient u konzumenta zprávy registrovat tzv. posluchače zpráv (*message listener*). Jakmile zpráva dorazí na místo určení, JMS poskytovatel ji doručí příjemci, přičemž k její obsluze vyvolá registrovaného posluchače. Klient konzumenta s posluchačem zpráv není při čekání na zprávu blokován.

4.7 Programovací model JMS API

Tato kapitola popisuje hlavní rozhraní JMS API a jejich význam, vztahy a způsoby implementace v klientských aplikacích, jak je popisuje JMS specifikace.

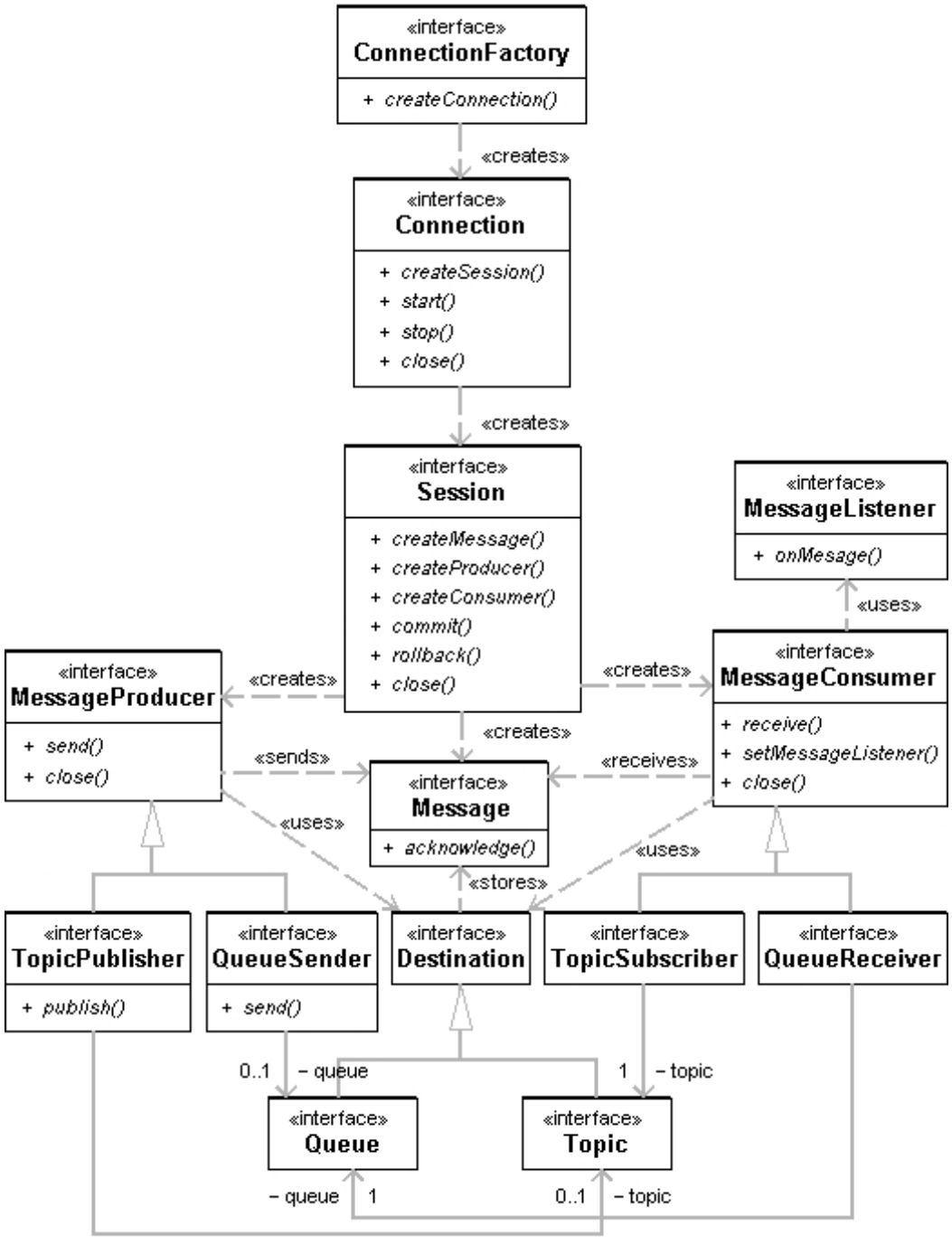
Veškerá rozhraní JMS API se na platformě Java EE nacházejí v balíčku javax.jms. Základními elementy tohoto balíčku jsou rozhraní administrovaných objektů továren spojení a míst určení (ConnectionFactory a Destination), objektů spojení (Connection), objektů relací (Session), producentů a konzumentů zpráv (MessageProducer a MessageConsumer) a entit vlastních zpráv (Message). Vztahy mezi těmito rozhraními zachycuje UML třídní diagram na obrázku 7.

Jak lze z diagramu částečně vyčíst, JMS API obsahuje kromě obecných rozhraní i rozhraní specifická pro obě messaging domény (PTP i Pub/Sub), avšak použití rozhraní společných pro oba modely, umožňuje implementovat JMS aplikace jednotným způsobem bez ohledu na doménu nasazení.[11] V tabulce 3 jsou zobrazeny vztahy mezi rozhraními obou komunikačních domén.

4.7.1 Administrované objekty

Místa určení a továrny spojení jsou dvě části aplikace, které spíše je vhodné spravovat administrativně než aplikačně. Technologie, na nichž jsou tyto objekty založené se s velkou pravděpodobností velice liší od jedné implementace JMS specifikace k druhé, a proto správa a konfigurace těchto objektů patří spolu s dalšími administrativními úkoly k operacím, které se různí od poskytovatele k poskytovateli.

[11] Následující ukázky Java kódu pro implementaci JMS klientů vždy používají obecná rozhraní, takže vytvoření klienti mohou sloužit pro oba komunikační modely. Konkrétní doména je určena skutečným typem objektu továrny spojení a místem určení (tj. zda se jedná o frontu, či téma zpráv).



Obrázek 7: Třídní digram základních rozhraní JMS API

Společná rozhraní	PTP rozhraní	Pub/Sub rozhraní
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

Tabulka 3: Vztahy mezi rozhraními PTP a Pub/Sub domény

JMS klienti obvykle získávají tyto objekty z JNDI registru prostřednictvím JNDI API a poté k nim přistupují přes rozhraní JMS API. Obě uvedené API specifikace jsou přenositelné, takže klientská aplikace může běžet takřka beze změny nad různými implementacemi poskytovatelů JMS.

Továrny spojení

Továrna spojení (*connection factory*) je administrovaný objekt, který klient používá pro tvorbu spojení na JMS poskytovatele. Továrna spojení zapouzdřuje sadu konfiguračních parametrů a nastavení definovaných administrátorem pro vytváření spojení na JMS poskytovatele. Pozoruhodné je, že ačkoliv je továrna spojení nedílnou součástí JMS, tak JMS specifikace nedefinuje, jaké informace by měla továrna spojení zapouzdřovat.[1] Obvykle to však jsou údaje o serveru, na němž běží JMS poskytovatel, a portu, kde naslouchá. JMS také nestandardizuje způsob, jakým klient získává objekt továrny spojení od JMS poskytovatele. Většinou to však bývá pomocí výše zmiňované JNDI služby a to buď ve vlastní režii prostřednictvím JNDI API, nebo s využitím vstřikování závislých zdrojů (*dependency injection*) na Java EE aplikačních serverech. Oba postupy lze vidět na následujícím výpisu kódu 1.

```
// Vyhledání pomocí JNDI služby
Context ctx = new InitialContext() ; // tvorba kontextu hledání v JNDI, parametry načteny ze
    souboru jndi.properties
ConnectionFactory cf = (ConnectionFactory) ctx.lookup(CONN_FACTORY_NAME); // vyhledání
    objektu továrny spojení
// Vyhledání s využitím vstřikování závislých zdrojů
@Resource(mappedName=CONN_FACTORY_NAME) // specifikace JNDI jména vstřikovaného
    objektu
private static ConnectionFactory cf; // o naplnění proměnné se postará aplikační server
```

Výpis 1: Získání objektu továrny spojení

Továrna spojení je podle JMS specifikace definována jako rozhraní ConnectionFactory poskytující metody pro tvorbu objektů spojení (createConnection). V případě požadavku vytvářet spojení na fronty zpráv je objekt továrny spojení instancí specifičtějšího typu QueueConnectionFactory, resp. TopicConnectionFactory, je-li potřeba připojovat se na témata zpráv.

Místa určení

Místo určení (*destination*) reprezentované instancí obecného rozhraní Destination je objekt, který klient používá pro určení cíle zpráv, které vyprodukuje, či zdroje zpráv, které zkonzumuje. V PTP doméně se místa určení nazývají fronty zpráv (rozhraní Queue). V doméně Pub/Sub jsou místa určení označovány jako témata zpráv (rozhraní Topic).

Místo určení zapouzdřuje adresu specifickou pro poskytovatele, spolu s dalšími konfiguračními informacemi, avšak JMS specifikace nedefinuje standard formátu adresy. Fyzické umístění místa určení na serveru je určeno poskytovatelem. JMS aplikace může používat vícero front a témat zpráv v závislosti na požadavcích.[1]

Objekt místa určení klient obvykle získává opět z JNDI služby (viz zdrojový kód 2).

```
// Vyhledání pomocí JNDI služby
Context ctx = new InitialContext() ; // tvorba JNDI kontextu hledání
Destination destination = (Destination) ctx.lookup(DESTINATION_NAME); // vyhledání objektu
    místa určení
// Vyhledání s využitím vstřikování závislých zdrojů
@Resource(mappedName=DESTINATION_NAME) // specifikace JNDI jména vstřikovaného objektu
private static Destination destination; // proměnná, jenž má být naplněna
```

Výpis 2: Získání objektu místa určení

4.7.2 Spojení

Spojení (*connection*) zapouzdřuje logické propojení na JMS poskytovatele. Na spojení lze pohlížet jako na komunikační kanál mezi aplikací a messaging serverem. Spojení může být kupříkladu reprezentováno otevřeným TCP/IP socketem mezi klientem a službou poskytovatele. Jako na analogii JMS spojení lze pohlížet na připojení k hlavní telefonní lince, která propojuje klientské telefonní přístroje s ústřednou.

Jak bylo uvedeno výše, spojení jsou vytvářena pomocí metod továren spojení. Spojení jsou dále používána pro tvorbu jedné či více relací pro zasílání a příjem zpráv do míst určení, resp. z nich.

V JMS specifikaci jsou spojení definována jako instance rozhraní Connection, respektive některého z jeho dvou podtypů — QueueConnection či TopicConnection -– v závislosti na messaging doméně.[1]

Při vytváření spojení dochází, mimo jiné, k autentizaci klientů a k nastavování parametrů komunikace. Nebot' udržování spojení není triviální úkol, jsou instance spojení poměrně komplikovanými a robustními objekty, které využívají mnoho systémových zdrojů, jež je nutno po ukončení práce uvolnit k použití dalšími službami, a tedy spojení řádně uzavřít (metodou close). Uzavřením spojení se také uzavřou všechny jeho aktivní relace a vytvoření producenti a konzumenti zpráv.

Po vytvoření se spojení nachází v tzv. zastaveném (*stopped*) režimu, po jehož dobu trvání nemohou být přijaty žádné zprávy (odesílání zpráv je však možné). Toto opatření umožňuje klientům zabývat se přípravou pracovního prostředí, aniž by byli rušeni nut-

ností zpracovávat příchozí zprávy, dokud sami nebudou připraveni. K aktivaci spojení, a tedy i k uvolnění blokace přijímání zpráv je nutné zavolat metody start.[12]

Práci s objektem spojení (vytvoření, aktivaci a uzavření) ukazuje následující zdrojový kód 3.

```
Connection conn = cf.createConnection(); // tvorba spojení továrnou spojení
try {
    ... // nastavení pracovního prostředí, tvorba relace, konzumentů a producentů
    conn.start() ; // aktivace spojení pro příjem zpráv
    ... // tvorba, příjem a odesílání zpráv
} finally {
    conn.close(); // uzavření spojení a uvolnění všech zdrojů
}
```

Výpis 3: Vytvoření, aktivace a uzavření spojení

4.7.3 Relace

Relace (*session*) je jednovláknovým kontextem pro produkci a konzumaci zpráv. Relace vytváří nad JMS spojením rámec, v němž dochází ke komunikaci (výměně zpráv) mezi klientem a poskytovatelem.

Relace se používají pro tvorbu producentů a konzumentů zpráv a především pro vytváření zpráv samotných. Relace zprostředkovávají JMS klientům přístup ke spojení za účelem posílání a příjmu zpráv.

Relace poskytují transakční kontext, jímž lze seskupit sadu požadavků na odeslání či příjem zpráv do atomické (nedělitelné) jednotky práce, která bude provedena jako jeden celek. Kontext odkládá zprávy určené k doručení, dokud není transakce potvrzena (metodou commit), teprve poté jsou zprávy doručeny. Před tím, než je transakce potvrzena, může uživatel zrušit doručení zpráv jejím odvoláním (metoda rollback).[1]

Používání transakcí je nepovinné, standardně dochází k pokusu o doručení zprávy v okamžiku, kdy je odeslána. Konzumenti proto musí odesílat JMS poskytovateli zpět potvrzení o úspěšném přijetí zprávy (metoda acknowledge rozhraní Message), čímž zabrání pokusu o opětovné doručení zprávy.

Relace je klientův privátní náhled na spojení. Každé spojení může mít ve stejný čas více aktivních relací. Podobně jako je spojení nutné pro komunikaci s JMS poskytovatelem, tak relace je nezbytná pro komunikaci se spojením. Analogií relace lze v doméně telekomunikací spatřovat v konkrétním telefonním hovoru, uskutečněném prostřednictvím telefonní linky.

Obecné metody pro JMS relace jsou specifikovány v rozhraní Session. V podtypech QueueSession, resp. TopicSession nalezneme metody specifické pro určitou messaging doménu.

Následující výpis zdrojového kódu zachycuje základní průběh práce s objektem relace při požadavku na transakční zpracování.

[12] Doručování zpráv lze opětovně inhibovat metodou stop.

```
Session session = conn.createSession(true, Session.SESSION_TRANSACTED); // vytvoření relace
    pro transakční zpracování
try {
    ... // tvorba producentů a konzumentů
    session.commit(); // potvrzení transakce
} catch (Exception e) {
    session.rollback() ; // zrušení (odvolání) transakce v případě výskytu výjimky
} finally {
    session.close() ; // uzavření relace a uvolnění všech jejích zdrojů
}
```

Výpis 4: Vytvoření transakční relace

Prostřednictvím relace samotné lze sice vytvářet zprávy, avšak nelze je ani posílat, ani přijímat. K tomuto účelu slouží producenti, resp. konzumenti zpráv. Relace sehrává pouze roli továrny, která vytváří instance těchto objektů.

4.7.4 Producenti zpráv

Podle JMS specifikace je producentem zprávy (*message producer*) objekt vytvořený relací, jehož účelem je odesílání zpráv na místo určení.[1] Rozhraní MessageProducer obsahuje obecné metody pro odesílání zpráv bez ohledu na messaging doménu (varianty metody send). Metody specifické pro PTP doménu jsou definovány rozhraním QueueSender. V Pub/Sub doméně jsou producenti zpráv instancemi rozhraní TopicPublisher.

Způsob vytvoření a odeslání zprávy ukazuje následující výpis kódu.

```
MessageProducer producer = session.createProducer(destination); // tvorba producenta pro dané
    místo určení
try {
    Message message = ... // vytvoření požadovaného typu zprávy
    producer.send(message); // odeslání dříve vytvořené zprávy
} finally {
    producer.close(); // uvolnění zdrojů producenta zpráv
}
```

Výpis 5: Vytvoření producenta zpráv

4.7.5 Konzumenti zpráv

JMS specifikace definuje konzumenta zpráv (*message consumer*) jako objekt vytvořené relací, jejichž účelem je přijímat zprávy zaslané na místo určení.[1] Prostřednictvím konzumentů zpráv registruje klient u JMS poskytovatele svůj zájem o příjem zpráv z určitého místa určení. JMS poskytovatel pak sám řídí doručování zpráv z míst určení k jejím registrovaným konzumentům.

Konzumaci zpráv nezávislou na doméně nabízí rozhraní MessageConsumer. Konzumenti zpráv v PTP doméně implementují rozhraní QueueReceiver, v doméně Pub/Sub se jedná o rozhraní TopicSubscriber.

V obou typech messaging domén může konzument přijímat zprávy synchronním i asynchronním způsobem. K synchronnímu (blokujícímu) přijmu zprávy slouží metoda receive. Pro asynchronní (neblokující) příjem zpráv se využívá objektu posluchače zpráv (*message listener*), kterého je třeba registrovat u konzumenta zprávy.

Zdrojový kód 6 ukazuje způsob vytvoření konzumenta zpráv a synchronní příjem zprávy.

```
MessageConsumer consumer = session.createConsumer(destination); // tvorba konzumenta zpráv
    z daného místa určení
try {
    conn.start() ;  // aktivace spojeni
    Message message = consumer.receive(); // přijmutí zprávy
    ... // zpracování zprávy
    message.acknowledge(); // potvrzení úspěšného přijetí zprávy
} finally {
    consumer.close(); // uvolnění zdrojů konzumenta zpráv
}
```

Výpis 6: Vytvoření konzumenta zpráv

Posluchači zpráv

Podle JMS je posluchač zpráv (*message listener*) objekt, který vystupuje jako asynchronní obsluha události doručení zprávy.[1] Objekty posluchačů zpráv implementují rozhraní MessageListener obsahující jedinou metodu onMessage, v níž se definují akce pro zpracování příchozí zprávy. Konzument zpráv registruje posluchače zpráv prostřednictvím metody setMessageListener.[1]

Každá relace zajišťuje sériové předání přijatých zpráv registrovanému posluchači, takže posluchač přiřazen k jednomu či více konzumentům stejné relace může zpracovávat v daný okamžik vždy jen jednu zprávu.

Posluchači zpráv se nerozlišují podle messaging domény. Stejný posluchač může přijímat zprávy jak z fronty, tak i z tématu zpráv v závislosti na tom, zda byl posluchač registrován konzumentem typu QueueReceiver, nebo TopicSubscriber.

Od verze 1.3 platformy J2EE byl zaveden speciální typ EJB komponent tzv. zprávami řízené beany (*Message Driven Beans*, zkráceně *MDB*). Tyto komponenty fungují jako posluchači zpráv v rámci EJB kontejneru J2EE serveru. Zprávami řízené beany jsou podrobněji popsány v kapitole 4.11.2.

4.7.6 Zprávy

Zpráva (*message*) je objekt, jenž v rámci JMS API implementuje rozhraní Message či některý z jeho podtypů a který představuje vlastní předmět komunikace JMS klientů. Jedná se tedy o entitu předávanou mezi softwarovými aplikacemi prostřednictvím JMS poskytovatele. Ačkoliv je formát JMS zpráv vcelku jednoduchý, tak je zároveň velice flexibilní a dovoluje vytvářet zprávy ve formátu kompatibilním s aplikacemi jiných platforem i bez podpory JMS.[9]

Zpráva se skládá ze tří hlavních částí – hlavičky, sady vlastností a těla zprávy. Detailnější popis struktury JMS zpráv následuje v kapitole 4.8.

4.7.7 Souběžné použití

Vzhledem k tomu, že podpora souběžného přístupu ke zdrojům sebou typicky nese další režii a složitost, některá z uvedených rozhraní nebyla navržena tak, aby umožňovala souběžný chod. Ten se omezuje pouze na instance rozhraní určené pro sdílení vícevláknovými klienty (Destination, ConnectionFactory, Connection), ostatní rozhraní jsou navržena tak, aby k jejich instancím přistupovalo v jeden okamžik pouze jedno vlákno (Session, MessageProducer a MessageConsumer.

Pro omezení souběžného použití objektů relací jsou především dva důvody. Předně jsou JMS relace těmi entitami, které podporují transakční zpracování. Je velice obtížné implementovat transakce, které jsou vícevláknové. Za druhé relace podporují asynchronní konzumaci zpráv, což znamená, že JMS nevyžaduje, a neumožňuje, aby při asynchronní konzumaci zpráv bylo možno zpracovávat více zpráv souběžně. Souběžnost lze získat podle potřeby použitím více objektů relací.[1]

4.7.8 Ošetřování výjimek

K důležitým prvkům tvorby robustních aplikací patří samozřejmě důsledné ošetřování výjimek. U messaging systémů a distribuovaných systémů obecně to platí dvojnásob. Proto je nutno představit způsoby ohlašování chybových stavů JMS aplikací.

JMS API definuje výjimku typu JMSException jako kořenovou třídu všech výjimek vyvolaných JMS metodami. Jedná se o kontrolovanou (checked) výjimku, jejíž odchytávání poskytuje obecný způsob zpracování všech výjimek majících vazbu na JMS.

Standardní JMS výjimky

Vedle JMSException definuje JMS specifikace několik dalších typů výjimek, které standardizují způsob hlášení chybových stavů.[1]

- IllegalStateException – při volání metod objektů nacházejících se v nepatřičném stavu
- JMSSecurityException – v případě odmítnutí operace z důvodu bezpečnostních omezení
- InvalidClientIDException – při pokusu o chybné nastavení klientského identifikátoru
- InvalidDestinationException – v případě, je-li specifikováno neplatné místo určení
- InvalidSelectorException – při chybné syntaxi selektoru zpráv
- MessageEOFException – při pokusu o čtení obsahu zprávy, ačkoliv již byl dosažen konec toku dat
- MessageFormatException – v případě čtení či zápisu chybného, nebo nepodporovaného datového typu
- MessageNotReadableException – při pokusu o čtení zprávy určené pouze k zápisu (write-only)

- MessageNotWriteableException – při pokusu o zápis do zprávy určené pouze ke čtení (read-only)
- ResourceAllocationException – v případě, není-li poskytovatel schopen vyhradit pro požadovanou operaci zdroje
- TransactionInProgressException – při pokusu o provedení nepovolené operace při aktivní transakci
- TransactionRolledBackException – při neočekávaném odvolání transakce

Posluchači výjimek

Kromě klasického způsobu odchytu výjimek konstrukcí try–catch, umožňuje JMS API odchytávat výjimky při výskytu potíží se spojením i asynchronním způsobem za pomoci objektu typu ExceptionListener, kterého je třeba registrovat v objektu spojení.

Výjimky doručené posluchači výjimek jsou ty, jenž nemohou být jinak ohlášeny. ExceptionListener tedy neslouží k monitorování všech výjimek vyvolaných spojením.

4.8 Model JMS zpráv

Hlavním smyslem JMS aplikace je produkovat a konzumovat zprávy. Zprávy jsou středobodem messaging systémů, kolem něhož se vše točí. Proto je celá tato kapitola věnována právě problematice JMS zpráv.

Ačkoliv se definice zpráv mezi messaging systémy velmi liší, JMS poskytuje jednotný způsob, jak zprávy popisovat a jak k nim přistupovat (tj. formát jejich struktury a obsahu). Většina podnikových messaging produktů zachází se zprávami jako s jednoduchými (*lightweight*) entitami, které se skládají z hlavičky a užitečného obsahu (těla). Hlavička obsahuje položky potřebné pro směrování zprávy a její identifikaci, tělo obsahuje vlastní aplikační data, jenž jsou posílána.

Jak již bylo zmíněno, v rámci této obecné formy se definice zpráv významně liší napříč produkty. Hlavní rozdíly bývají v obsahu a sémantice hlaviček (nestrukturovaná data × kódovaná data, podpora úložišť pro definice typů zpráv). Pro JMS je tedy velice obtížným úkolem podchytit a podporovat celou šíři takovýchto modelů zpráv.[1]

Nyní již následuje samotný popis struktury JMS zpráv, které se podle JMS specifikace skládají z hlavičky, sady vlastností a těla.

4.8.1 Hlavička zprávy

Hlavička JMS zprávy (*message header*) obsahuje několik předdefinovaných položek, které představují informace používané klienty a poskytovateli pro identifikaci a směrování zpráv. Všechny JMS zprávy podporují stejnou sadu položek hlavičky. Například pro každou zprávu je definován jednoznačný identifikátor, časové razítko odeslání, či úroveň priority. Až na několik výjimek jsou položky hlavičky povinné, a musí mít tedy nastavenou hodnotu.

Kompletní hlavička zprávy (tj. všechny její položky) je odeslána všem přijímajícím klientům. JMS specifikace však nedefinuje položky hlavičky odesílané jiným (ne-JMS) klientům.

JMS API definuje pro každou položku hlavičky přístupové metody (tzv. gettery a settery). Hodnoty některých položek musí být nastaveny explicitně klientem, ale mnohé jsou nastavovány automaticky producentem zprávy při jejím odesílání. Seznam položek hlavičky spolu s jejich typem a způsobem jejich nastavení je znázorněn v tabulce 4.

Položka hlavičky	Typ	Způsob nastavení
JMSDestination	Destination	producentem zprávy při odeslání (send či publish)
JMSDeliveryMode	int	producentem zprávy při odeslání (send či publish)
JMSExpiration	long	producentem zprávy při odeslání (send či publish)
JMSPriority	int	producentem zprávy při odeslání (send či publish)
JMSMessageID	String	producentem zprávy při odeslání (send či publish)
JMSTimestamp	long	producentem zprávy při odeslání (send či publish)
JMSRedelivered	boolean	JMS poskytovatelem při opětovném doručení
JMSCorrelationID	String	explicitně klientem (příslušným setterem)
JMSReplyTo	Destination	explicitně klientem (příslušným setterem)
JMSType	String	explicitně klientem (příslušným setterem)

Tabulka 4: Seznam položek hlavičky zpráv a způsob jejich nastavení

Následuje stručný popis každé položky hlavičky JMS zprávy.

JMSDestination

Tato položka obsahuje místo určení (frontu nebo téma zpráv), na nějž je zpráva posílána (resp. odkud byla přijata). Její hodnota je nastavována producentem při odeslání zprávy (původní hodnota je ignorována).

JMSDeliveryMode

Tato položka určuje doručovací režim zprávy, čímž ovlivňuje způsob, jakým JMS poskytovatel zajišťuje její doručení (zda ji má uchovávat, či nikoliv). O nastavení této položky se opět stará producent zpráv. Režimy doručování jsou podrobněji popsány v kapitole 4.10.3.

JMSMessageID

Hodnota této položky jednoznačně identifikuje každou odeslanou zprávu. Hodnotou JMSMessageID je řetězec, který působí v roli unikátního klíče identifikujícího zprávu v úložišti zpráv. Přesný rozsah unikátnosti takového klíče závisí na definici poskytovatele. Minimálně by měla pokrývat všechny zprávy konkrétní instalace JMS poskytovatele.

JMSTimestamp

Položka JMSTimestamp obsahuje časové razítko okamžiku, kdy byla zpráva předána poskytovateli k odeslání. Nejedná se tedy o dobu, kdy byla zpráva skutečně odeslána, neboť k tomu může dojít později (např. v důsledku transakčního zpracování či hromadění zpráv na klientské straně).

Při požadavku o odeslání zprávy (voláním metody send) je případná hodnota této položky ignorována. Po odeslání zprávy producentem obsahuje časový údaj z období mezi voláním a návratem odesílací metody.

JMSCorrelationID

Klient může využít tuto položku pro vytvoření vazby mezi zprávami. Typickým použitím je vazba zprávy odpovědi na zprávu požadavku, jenž ji vyvolal (princip požadavek-odpověď).

JMSCorrelationID smí obsahovat buď identifikátor zprávy specifický pro daného poskytovatele (tedy hodnoty z JMSMessageID), nebo libovolný řetězec příslušející dané aplikaci, ale taky pole nativních bajtů poskytovatele.

Nejčastěji je využívána první možnost, ale v některých případech, kdy je aplikace složena z více klientů, je vhodné provázat zprávy příslušející dané aplikaci prostřednictvím nějakého řetězce. Třetí možnost je podporována pouze některými poskytovateli za účelem podpory použití nativních klientů poskytovatele. Použití pole bajtů jako hodnoty pro JMSCorrelationID způsobí nepřenositelnost JMS aplikace mezi různými poskytovateli.

JMSReplyTo

Položka hlavičky JMSReplyTo obsahuje místo určení dodané producentem zprávy, na něž by měla být odeslána odpověď. Zprávy s vyplněnou hodnotou této položky reprezentují obvykle požadavky očekávající vydání odpovědi (princip požadavek-odpověď). Vydání odpovědi není však povinné. Rozhodnutí záleží na klientovi.

Naproti tomu zprávy, které tuto položku nemají vyplněnu, reprezentují typicky oznámení o různých událostech, nebo prostě obsahují nějaká data, jenž odesílatel považuje za důležitá.

JMSRedelivered

Pokud klient obdrží zprávu s nastavenou hodnotou této položky je pravděpodobné (nikoliv však jisté), že tato zpráva mu byla doručena již dříve, avšak dosud nebylo potvrzeno její přijetí. Obecně platí, že poskytovatel nastaví tuto položku v hlavičce zprávy, kdykoliv se snaží zprávu znovu doručit. Konzumující aplikaci je tak signalizováno, že tato zpráva již mohla být doručena dříve, a tedy že by aplikace měla přijmout mimořádná opatření, aby předešla duplicitnímu zpracování téže zprávy.

JMSType

Položka JMSType obsahuje identifikátor typu zprávy, dodaný klientem při jejím odeslání. Někteří JMS poskytovatelé používají tuto položku jako odkaz do vlastního úložiště definic typů zpráv. JMS nedefinuje standard pro definování úložišť definic typů zpráv, ani způsob pojmenování definic, jež obsahuje. Tato položka je vyžadována pouze některými poskytovateli. Nastavení hodnoty této položky je prováděno automaticky producentem zprávy při jejím odeslání bez ohledu na to, zda ji aplikace potřebuje ke své činnosti, či nikoliv.

JMSExpiration

Tato položka obsahuje okamžik vypršení platnosti (expirace) zprávy. Její hodnota je spočítána při odeslání zprávy jako součet uvedené hodnoty délky života (platnosti) zprávy (tzv. *time-to-live*, zkráceně *TTL*) a aktuálního času. Je-li TTL specifikováno jako 0, je tato položka nastavena také na hodnotu 0, čímž se indikuje, že zpráva nikdy nevyprší (tj. že má neomezenou platnost).

Když je při pokusu o doručení zatím nedoručené zprávy aktuální čas větší, než je vypočtená hodnota položky JMSExpiration, je zpráva zničena. Klienti by neměli přijímat zprávy, jimž vypršela platnost. Nicméně JMS specifikace nezajišťuje, že tento případ nenastane.

Další informace o problematice expirace zpráv jsou uvedeny v kapitole 4.10.4.

JMSPriority

Tato položka obsahuje úroveň priority doručení zprávy. Její hodnota se nastavuje prostřednictvím metody pro odeslání zprávy.

JMS specifikace definuje desetistupňovou škálu úrovní priority, kde hodnota 0 označuje nejnižší úroveň a 9 nejvyšší. Standardní (implicitní) hodnotou je úroveň 4. Zprávy s prioritou v rozmezí 0–4 se označují jako běžné zprávy (*normal messages*) a zprávy s prioritou 5–9 jako spěšné (*expedited*).

JMS specifikace sice striktně nevyžaduje, aby poskytovatelé implementovali řazení a doručování zpráv podle jejich priority, avšak očekává, že expresní zprávy budou doručeny před běžnými zprávami.

4.8.2 Vlastnosti zprávy

Na vlastnosti JMS zprávy (*message properties*) lze pohlížet jako na doplňkové položky hlavičky zprávy, tedy jako na jakýsi mechanismus pro přidání nepovinných položek k hlavičce zprávy. Jedná se většinou o položky specifické pro aplikaci, poskytovatele anebo o jiná nepovinná nastavení.

Tyto hodnoty bývají používány pro zajištění kompatibility s jinými messaging systémy nebo je může klient využít prostřednictvím tzv. selektorů zpráv (*message selectors*) k filtrování přijímaných zpráv na základě specifikovaných kritérií (selektory zpráv jsou podrobněji rozebrány v kapitole 4.10.1).

Vlastnosti mohou být trojího druhu: vlastnosti specifické pro aplikaci (uživatelské vlastnosti), vlastnosti specifické pro poskytovatele a předdefinované vlastnosti. JMS specifikace definuje názvy a významy vlastností posledně jmenované kategorie stejně jako způsob pojmenování vlastností specifických pro poskytovatele. Pojmenování vlastností specifických pro aplikaci není nijak definováno (kromě podmínky, aby nezačínali řetězcem „JMS“), a je tedy plně v kompetenci uživatele.

Názvy všech vlastností obecně však musí odpovídat pravidlům pro identifikátory selektorů zpráv (tzn. víceméně pravidlům pro identifikátory podle standardu SQL92). Vlastnost může nabývat hodnoty typu boolean, byte, short, int, long, float, double a String. Hodnoty vlastností lze vzájemně převádět. Podporované konverze jsou vyznačeny v ta-

bulce 5. S hodnotami vlastností lze rovněž pracovat objektově prostřednictvím příslušných obalových typů.

	boolean	byte	short	int	long	float	double	String
boolean	✓							✓
byte		✓	✓	✓	✓			✓
short			✓	✓	✓			✓
int				✓	✓			✓
long					✓			✓
float						✓	✓	✓
double							✓	✓
String	✓	✓	✓	✓	✓	✓	✓	✓

Tabulka 5: Podporované konverze hodnot vlastností zprávy („fajfka“ označuje povolenou konverzi hodnoty zapsané jako typ daný řádkem a čtené jako typ daný sloupcem)

Předdefinované vlastnosti zpráv

JMS specifikace definuje sadu vlastností, pro než vyhrazuje název začínající vždy prefixem „JMSX“. Jejich přítomnost v určité zprávě závisí na poskytovateli. Ten je může na základě administračních či jiných kritérií do některých zpráv zahrnout a v jiných je vynechat. Množinu JMSX vlastností podporovaných daným spojením lze zjistit metodou getJMSXPropertyNames rozhraní ConnectionMetaData.

Některé z předdefinovaných vlastností zpráv jsou nastavovány automaticky JMS poskytovatelem, jiné jsou určeny k nastavení klientem. Následuje jejich stručný popis.

JMSXUserID – řetězec (String) nastavovaný poskytovatelem při odeslání zprávy, který představuje identifikátor uživatele odesílajícího zprávu.

JMSXAppID – řetězec nastavovaný rovněž poskytovatelem při odeslání zprávy, jenž reprezentuje identifikátor aplikace odesílající zprávu.

JMSXDeliveryCount – hodnota typu int nastavovaná poskytovatelem při přijetí zprávy vyjadřující počet pokusů o doručení zprávy.

JMSXGroupID – řetězec, kterým klient identifikuje skupinu zpráv, jejíž součástí je tato zpráva.[13]

JMSXGroupSeq – hodnota typu int, kterým klient určuje pořadí zprávy v rámci skupiny zpráv.[13]

JMSXProducerTXID – řetězec nastavovaný poskytovatelem při odeslání zprávy, jenž slouží jako identifikátor transakce, v jejímž rámci byla zpráva vyprodukována.

[13] JMSXGroupID a JMSXGroupSeq představují standardní vlastnosti, podporované všemi poskytovateli, jenž by klient měl využít, pokud potřebuje seskupovat zprávy.

JMSXConsumerTXID – řetězec nastavovaný poskytovatelem při přijetí zprávy, jenž slouží jako identifikátor transakce, v jejímž rámci byla zpráva zkonzumována.

JMSXRcvTimestamp – hodnota typu long nastavovaná poskytovatelem při přijetí zprávy vyjadřující čas, kdy byla zpráva doručena konzumentovi.

JMSXState – hodnota typu int nastavovaná poskytovatelem reprezentující stav, v němž se zpráva nachází.[14]

Vlastnosti zpráv specifické pro poskytovatele

Pro názvy vlastností specifických pro JMS poskytovatele vyhrazuje JMS specifikace prefix „JMS_<poskytovatel>“, kde řetězec <poskytovatel> představuje hodnotu příslušející každému poskytovateli (kupř. messaging systém HornetQ používá řetězec „HQ“).

Smyslem vlastností specifických pro poskytovatele je zajistit podporu JMS při použití nativních klientů poskytovatele. Neměly by být používány pro předávání zpráv prostřednictvím JMS API.[1]

4.8.3 Tělo zprávy

Tělo zprávy představuje vlastní obsah zprávy, tj. užitečná data zprávy. JMS specifikace definuje pět formátů těla zpráv (také označované jako typy zpráv), což umožňuje předávat si data v mnoha odlišných formách a zároveň poskytuje prostředky k zajištění kompatibility mezi formáty zpráv existujících systémů.

JMS API definuje pro každý z těchto typů zvláštní rozhraní vycházející z rozhraní Message (viz obrázek 8), nedefinuje však způsob jejich implementace, pouze prostředky pro jejich vytváření a plnění daty.

Následuje stručný popis jednotlivých typů JMS zpráv.

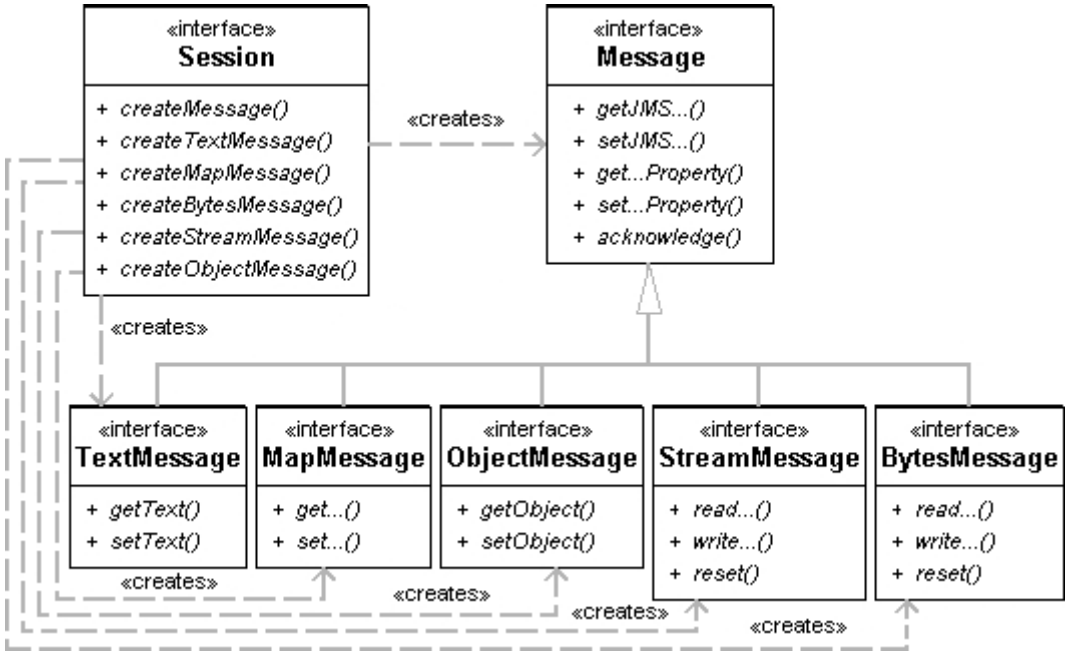
Textové zprávy

Zprávy typu TextMessage slouží pro přenos textových dat, a tudíž je jejich tělo tvořeno jednoduchým řetězcem v podobě objektu String. Tento typ zprávy je vhodný v situacích, kdy není třeba přenášet speciálně strukturovaná data, ale pouze prostý text. Populárním způsobem reprezentace obsahu textových zpráv je XML.

Mapové zprávy

Těla zpráv typu MapMessage obsahují množinu dvojic (neboli mapování) klíč-hodnota. Klíči jsou řetězce typu String a hodnotami jsou primitivní datové typy jazyka Java (a navíc ještě řetězce či pole bajtů). K záznamům lze přistupovat přímo prostřednictvím klíče nebo sekvenčně procházením všech mapování, avšak bez definovaného pořadí. Mapové zprávy podporují konverze podle tabulky 6.

[14] Stav zprávy může nabývat hodnoty: 1 – čekající (waiting), 2 – připravená (ready), 3 – prošlá (expired), nebo 4 – uchovaná (retained). Tato informace však vůbec nemá pro producenty, ani konzumenty význam, takže JMS neposkytuje pro vyhodnocování stavu zprávy žádné API.



Obrázek 8: Třídní digram typů JMS zpráv

Bajtové zprávy

Zprávy typu BytesMessage obsahují proud neinterpretovaných bajtů. Tento typ se používá při potřebě přenášet data v nativním formátu aplikace, který nemusí vyhovovat žádnému existujícímu JMS typu zprávy. Ve většině případů by však mělo být možno použít místo tohoto typu zprávy jeden z dalších typů. Interpretaci bajtů provádí příjemce s využitím běžných konverzí jazyka Java.

Ačkoliv JMS specifikace dovoluje používat vlastnosti i u bajtových zpráv, obvykle tak není činěno, neboť zahrnutí vlastností do zprávy může ovlivnit její formát.

Proudové zprávy

Zprávy typu StreamMessage jsou používány pro zasílání proudu hodnot primitivních datových typů programovacího jazyka Java. Primitivní typy jsou zapisovány, resp. čteny sekvenčně. Proudové zprávy jsou podobné zprávám bajtovým, avšak liší se způsobem konverze datových typů, jenž odpovídají tabulce 6.

Objektové zprávy

Obsahem těla zprávy typu ObjectMessage je libovolný serializovatelný Java objekt. Tímto typem zpráv je možno poslat i soubor více serializovatelných objektů a to prostřednictvím některé ze standardních kolekcí z balíčku java. util .

	boolean	byte	short	char	int	long	float	double	String	byte[]
boolean	✓								✓	
byte		✓	✓		✓	✓			✓	
short			✓		✓	✓			✓	
char				✓					✓	
int					✓	✓			✓	
long						✓			✓	
float							✓	✓	✓	
double								✓	✓	
String	✓	✓	✓		✓	✓	✓	✓	✓	
byte										✓

Tabulka 6: Podporované konverze u zpráv typu StreamMessage a MapMessage („fajfka“ označuje povolenou konverzi hodnoty zapsané jako typ daný řádkem a čtené jako typ daný sloupcem)

Prosté zprávy

Jako na speciální typ zpráv můžeme pohlížet na instance obecného rozhraní Message. Takové zprávy nemají žádné tělo, pouze hlavičku a sadu vlastností zprávy. Tento typ zpráv je užitečný, když aplikace nevyžaduje tělo zprávy.

4.9 Jednoduchá JMS aplikace

Na základě výpisů kódu v kapitole 4.7 lze sestavit jednoduchou JMS aplikaci, kterou tvoří dva klienti – producent a konzument. Následující výpisy kódu ukazují způsob jejich implementace.

Za pozornost jistě stojí skutečnost, že jsou na všech místech použita rozhraní společná oběma doménám komunikace, a tedy nelze z kódu aplikace určit, zda je vytvořena pro PTP či Pub/Sub komunikační model. Vše se odvíjí od skutečného typu administrovaných objektů, jak je definoval administrátor.

Na výpisu kódu 7 je zachycena část aplikace společná oběma klientům, obsahující metody pro získání administrovaných objektů továrny spojení a místa určení z JNDI služby.

```
public class CommonUtils {
  // Jmenný kontext pro JNDI
  private static Context jndiContext = null;
  // Jméno, s nímž je svázán objekt typu QueueConnectionFactory
  public static final String QUEUE_CF_NAME = "QueueFactory";
  // Jméno, s nímž je svázán objekt typu TopicConnectionFactory
  public static final String TOPIC_CF_NAME = "TopicFactory";
  // Jméno, s nímž je svázán objekt typu Queue
  public static final String QUEUE_NAME = "/queue";
  // Jméno, s nímž je svázán objekt typu Topic
  public static final String TOPIC_NAME = "/topic";
```

```
    // Získání JNDI kontextu
    private static Context getJNDIContext() {
      if (jndiContext == null) {
        try {
          jndiContext = new InitialContext();
        } catch (NamingException e) {
          System.err.println("Nepodarilo se vytvorit JNDI kontext: " + e);
        }
      }
      return jndiContext;
    }
    // Získání tovarny spojení
    public static ConnectionFactory getConnectionFactory(String name) {
      try {
        return (ConnectionFactory) getJNDIContext().lookup(name);
      } catch (NamingException e) {
        System.err.println("Nepodarilo se ziskat tovarnu spojeni: " + e);
        return null;
      }
    }
    // Získání místa určení
    public static Destination getDestination(String name) {
      try {
        return (Destination) getJNDIContext().lookup(name);
      } catch (NamingException e) {
        System.err.println("Nepodarilo se ziskat misto urceni: " + e);
        return null;
      }
    }
  }
```

Výpis 7: Metody společné odesílateli i příjemci

4.9.1 Kód producenta zpráv

Výpis kódu 8 představuje klienta produkujícího zprávy, který vytvoří objekty spojení, netransakční relace a producenta zpráv, načež odešle na místo určení jednoduchou textovou zprávu a poté uzavře spojení a s tím i uvolní všechny zdroje.

```
public class SimpleProducer {
  // Při zadání nepovinného parametru 'T' bude klient vydavatelem v doméně Pub/Sub. Implicitně
      se chová jako odesílatel PTP domény.
  public static void main(String[] args) {
    String connFactName = CommonUtils.QUEUE_CF_NAME;
    String destinationName = CommonUtils.QUEUE_NAME;
    if ((args.length > 0) && args[0].equalsIgnoreCase("T")) {
      connFactName = CommonUtils.TOPIC_CF_NAME;
      destinationName = CommonUtils.TOPIC_NAME;
    }
    // získání administrovaných objektu
    ConnectionFactory connFact = null;
    Destination destination = null;
```

```
    try {
      connFact = CommonUtils.getConnectionFactory(connFactName);
      destination = CommonUtils.getDestination(destinationName);
    } catch (Exception e) {
      System.err.println("Nepodarilo se ziskat admin. objekty: " + e);
      System.exit(1);
    }
    // vytvoření spojení, relace, producenta, odeslání zprávy a uvolnění zdrojů
    Connection connection = null;
    try {
      connection = connFact.createConnection();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageProducer producer = session.createProducer(destination);
      TextMessage message = session.createTextMessage("Ahoj, svete!");
      producer.send(message);
      System.out.println("Zprava odeslana: " + new Date());
    } catch (JMSException e) {
      System.err.println("Nepodarilo se odeslat zpravu: " + e);
    } finally {
      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {}
      }
    }
  }
}
```

Výpis 8: Klient produkující zprávy

4.9.2 Kód konzumenta zpráv

Klient na výpisu 9 slouží k synchronnímu příjmu textové zprávy. Nejprve opět vytvoří objekty spojení, netransakční relace a konzumenta zpráv, načež odblokuje spojení a čeká na příjem zprávy z místa určení (maximálně však 5 s). Poté uzavře spojení a s tím i uvolní všechny zdroje.

```
public class SimpleConsumer {
  // Při zadání nepovinného parametru 'T' bude klient odběratelem v doméně Pub/Sub. Implicitně se
       chová jako příjemce PTP domény.
  public static void main(String[] args) {
    String connFactName = CommonUtils.QUEUE_CF_NAME;
    String destinationName = CommonUtils.QUEUE_NAME;
    if ((args.length > 0) && args[0].equalsIgnoreCase("T")) {
      connFactName = CommonUtils.TOPIC_CF_NAME;
      destinationName = CommonUtils.TOPIC_NAME;
    }
    // získání administrovaných objektu
    ConnectionFactory connFact = null;
    Destination destination = null;
    try {
      connFact = CommonUtils.getConnectionFactory(connFactName);
```

```
        destination = CommonUtils.getDestination(destinationName);
      } catch (Exception e) {
        System.err.println("Nepodarilo se ziskat admin. objekty: " + e);
        System.exit(1);
      }
      // vytvoření spojení, relace, konzumenta, odeslání zprávy a uvolnění zdrojů
      Connection connection = null;
      try {
        connection = connFact.createConnection();
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageConsumer consumer = session.createConsumer(destination);
        connection.start(); // aktivace spojení
        Message messge = consumer.receive(5000); // čekání na zprávu
        if (message != null) {
          System.out.println("Zprava prijata: " + new Date());
          if (message instanceof TextMessage) {
            String content = ((TextMessage) message).getText();
            System.out.println("Obsah: " + content);
          } else {
            System.out.println("Nejedna se o textovou zpravu.");
          }
        } else {
          System.out.println("Zprava nebyla prijata.");
        }
      } catch (JMSException e) {
        System.err.println("Nepodarilo se prijmout zpravu: " + e);
      } finally {
        if (connection != null) {
          try {
            connection.close();
          } catch (Exception e) {}
        }
      }
    }
  }
```

Výpis 9: Klient konzumující zprávy

Když jsou uvedení klienti přeloženi a po předchozí správné konfiguraci prostředí postupně spuštěni (v pořadí SimpleProducer, SimpleConsumer), zobrazí konzument hlášení o přijetí zprávy podobné tomuto:

```
Zprava prijata: Fri Jul 16 11:12:00 CEST 2010
Obsah: Ahoj, svete!
```

4.10 Pokročilé mechanismy JMS

V této kapitole jsou popsány některé pokročilé vlastnosti JMS API potřebné pro tvorbu robustních a spolehlivých aplikací. Mnoho JMS aplikací si nemůže dovolit ztrácet či přijímat vícenásobně tutéž zprávu, a tak vyžadují, aby každá zpráva byla klientem přijata právě jednou. Uvedené koncepty umožňují docílit požadované úrovně spolehlivosti a výkonu aplikace.

4.10.1 Filtrování zpráv

JMS aplikace může filtrovat přijaté zprávy ve vlastní režii tak, že po přijetí zprávy klient sám určí na základě informací obsažených ve zprávě (většinou položek hlavičky či vlastností zpráv), zda chce zprávu dále zpracovat či nikoliv. Tento přístup však zbytečně zatěžuje klienta (i poskytovatele) nutností přijímat a zpracovat zprávy, o něž nemá zájem.

JMS nabízí mechanismus pro filtrování zpráv na straně poskytovatele, takže poskytovatel doručí konzumentovi pouze ty zprávy, o něž projevil zájem. K tomuto účelu slouží tzv. selektory zpráv (*message selector*). Jejich prostřednictvím konzument specifikuje kritéria (podmínky), která musí zpráva splňovat, aby mu mohla být doručena.

Tento způsob filtrace je postaven na podobném principu jako dotazování pomocí jazyka SQL. Selektor zpráv zde vystupuje jako řetězcem výrazu obsahujícího podmínky selekce. Zprávy lze filtrovat jak podle položek hlavičky zprávy, tak podle jejich vlastností. Konzument přijímá pouze ty zprávy, jejichž hlavička a vlastnosti vyhovují podmínkám specifikovaných selektorem. Prostřednictvím selektoru nelze filtrovat zprávy na základě obsahu jejich těl.

Syntaxe podmínek selektorů zpráv je založena na podmnožině standardu SQL92 pro syntaxi podmínek klauzule languageWHERE. Výraz selektoru může být libovolně složitý. Použité identifikátory musí vždy odpovídat názvům vlastností, či položkám hlavičky. Příklad selektoru zpráv může být řetězec language"JMSType = 'computer' AND processor IN ('Intel', 'AMD')AND (dualCore = TRUE OR memory $>$ 1024)".

Filtrace zpráv pomocí selektorů snižuje provoz na síti (méně častá komunikace, menší objem přenášených dat), avšak může zvýšit zatížení serveru (probíhá vždy na straně poskytovatele, vyhodnocení kritérií může být časově náročné).

Selektor zpráv konzumenta je specifikován při vytváření objektu MessageConsumer metodou Session.createMessageConsumer a zůstává stejný po celou dobu jeho existence.

4.10.2 Potvrzování zpráv

Potvrzování přijmu zpráv je v JMS implementováno na dvou úrovních – při zasílání zprávy JMS poskytovateli a při zasílání zprávy konzumentovi.

Při zasílání zprávy JMS poskytovateli, poskytovatel potvrdí doručení zprávy jejímu producentovi. Toto potvrzení příjetí je plně v kompetenci JMS poskytovatele a probíhá automaticky.

Při zasílání zprávy příjemci potvrzuje příjem zprávy sám konzument a navíc, aby se předešlo možným duplicitám, zasílá JMS poskytovatel opět automaticky potvrzení, že přijal potvrzení příjetí zprávy od konzumenta.

Dokud není zpráva potvrzena, není považována za úspěšně zkonzumovanou. Proces úspěšné konzumace zprávy běžně probíhá ve třech fázích:

1. Klient přijme zprávu.
2. Klient zpracuje zprávu.
3. Zpráva je potvrzena.

Zprávy přijaté v rámci transakce jsou potvrzovány automaticky při potvrzení transakce. Je-li transakce odvolána, jsou všechny zkonzumované zprávy opětovně doručeny.

V relacích bez transakčního zpracování je způsob a okamžik potvrzování přijatých zpráv určen tzv. potvrzovacím režimem (*acknowledgement mode*), který se stanovuje při vytvoření relace (Connection.createSession). JMS specifikace definuje tři typy těchto módů:

CLIENT_ACKNOWLEDGE – Tento mód určuje, že klient potvrzuje zprávu explicitně prostřednictvím metody Message.acknowledge. Moment potvrzení je tedy plně v kompetenci klienta. Potvrzením jediné zkonzumované zprávy se automaticky potvrdí i příjem všech zpráv doručených toutéž relací.

AUTO_ACKNOWLEDGE – V tomto módu jsou zprávy potvrzovány automaticky při svém přijetí (tj. při návratu z metody MessageConsumer.receive nebo MessageListener.onMessage). K potvrzování tedy dochází ještě před vlastním zpracováním zprávy.

DUPS_OK_ACKNOWLEDGE – Touto volbou se dává najevo, že konzumentům nevadí příjem duplicitních zpráv, takže potvrzení přijetí zprávy může být odesláno později. Klient potvrzuje zprávy po skupinách. Tímto lze redukovat režii relace omezením potřeby činit kroky nutné k předcházení duplicitám.

4.10.3 Režimy doručování zpráv

Režimy doručování zpráv (*delivery mode*) ovlivňují způsob, jakým poskytovatel nakládá se zprávami při jejich doručování, tj. zda budou perzistentní, či nikoliv. Perzistentní zprávy musí být uchovány způsobem, že nesmí být ztraceny ani v případě selhání systému poskytovatele. Doručovací režim se stanovuje nastavením požadované hodnoty položky metodou Message.setJMSDeliveryMode v hlavičce zprávy.

JMS specifikace definuje dva typy těchto režimů:

PERSISTENT – Tato volba nařizuje JMS poskytovateli, aby přijal opatření nutná k zaručení, aby žádná zpráva nebyla v případě selhání systému poskytovatele ztracena. V tomto módu jsou zprávy při svém odeslání zaznamenány do stabilního úložiště, kde přetrvají i případný pád systému. Zprávy odeslané v tomto režimu musí být doručeny právě a pouze jednou (tj. zpráva nesmí být ani ztracena, ani doručena dvakrát).

NON_PERSISTENT – Tento režim požaduje po JMS poskytovateli, aby uchovával zaslané zprávy, tj. negarantuje, že zprávy nebudou v případě selhání ztraceny. Tento režim je však spojen s nižší režií, a vylepšuje tedy výkon aplikace. Zprávy odeslané v tomto režimu jsou doručeny klientům nanejvýš jednou (tj. zpráva nesmí být doručena dvakrát, ale může být ztracena).

Implicitně je používáno perzistentní doručování. Použitím neperzistentního režimu doručování lze vylepšit výkon aplikace a snížit režii spojenou s ukládáním zpráv do úložiště, avšak toho lze využít pouze tehdy, když si aplikace může dovolit přicházet o zprávy.

4.10.4 Expirace zpráv

Aby se předešlo hromadění nedoručených a nedoručitelných zpráv ve frontách a tématech JMS poskytovatele, zavedla JMS specifikace mechanismus pro stanovení doby platnosti každé zprávy. Implicitně zprávám platnost nikdy nevyprší, což může v případě neúspěšných pokusů o jejich doručení vést k jejich hromadění u poskytovatele, a tedy i ke zvýšení režie spojené s jejich uchováváním a pokusy o jejich doručení, v krajním případě až k vyčerpání všech dostupných zdrojů (např. místa na disku).

Proto může producent zpráv určit každé zprávě prostřednictvím parametru time−to −live metody send dobu platnosti odesílané zprávy (z níž se následně odvodí hodnota hlavičky JMSExpiration), po jejímž uplynutí je zpráva poskytovatelem odstraněna z úložiště a nadále se poskytovatel již nepokouší o její doručení.

Způsob stanovení a výpočtu okamžiku vypršení platnosti zprávy je popsán v kapitole 4.8.1.

4.10.5 Dočasná místa určení

Ačkoliv většina poskytovatelů nabízí prostředky pro programové vytváření destinací, tak jsou obvykle fronty a témata zpráv definovány administrativně prostřednictvím dodávaných nástrojů, nebot'je očekáváno, že destinace jsou trvalého charakteru.

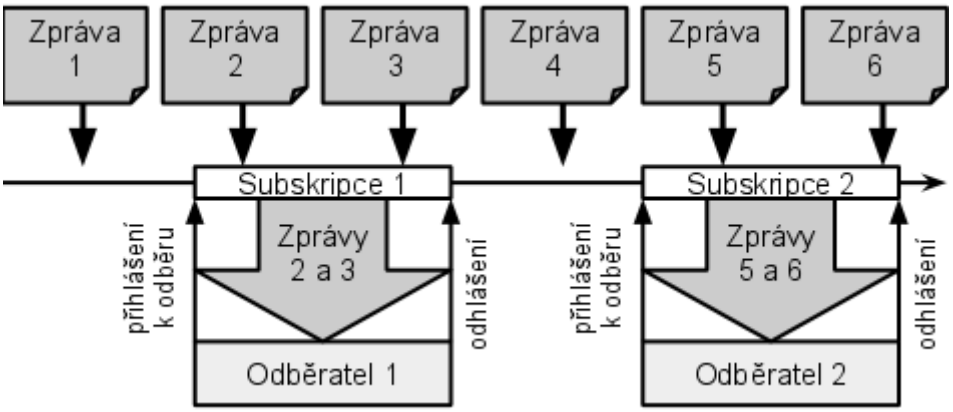
JMS API však dovoluje vytvářet místa určení dynamicky, které existují pouze po dobu trvání spojení, které je vytvořilo. Když je spojení uzavřeno, jsou tyto destinace odstraněny. Takové destinace jsou označovány jako dočasné (*temporary destinations*). K vytvoření dočasné fonty zpráv slouží metoda Session.createTemporaryQueue, dočasné téma se vytváří metodou Session.createTemporaryTopic.

Typickým použitím dočasných míst určení je implementace komunikace typu požadavek-odpověď. Při odeslání zprávy uvede odesílatel vytvořenou dočasnou destinaci jako hodnotu položky JMSReplyTo hlavičky zprávy, čehož může konzument využít jako místo určení, na než má zaslat odpověď, přičemž se na původní požadavek odkáže nastavením hlavičky JMSCorrelationID na hodnotu identifikátoru zprávy požadavku (položka JMSMessageID).

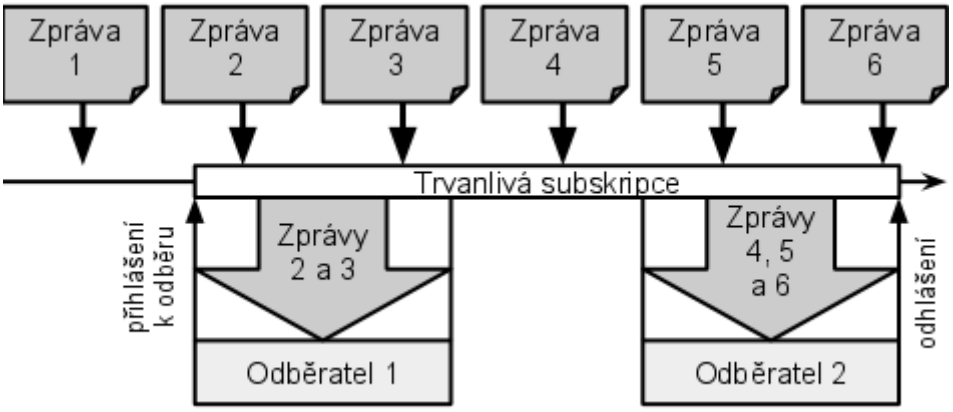
4.10.6 Trvanlivé subskripce

Při komunikaci typu publish/subscribe odběratel běžně přijímá pouze zprávy publikované v době, kdy je aktivní (viz obrázek 9).

Za cenu vyšší režie může být vytvořen odběratel s tzv. trvanlivou subskripcí (*durable subscription*), zkráceně označovaný jako trvanlivý odběratel (*durable subscriber*), který může přijmout zprávy publikované k tématu i v době, kdy nebyl aktivní (připojen). Trvanlivá subskripce má jednoznačnou identitu a může v čase registrovat více odběratelů, v jeden moment však má vždy maximálně jednoho odběratele. JMS uchovává zprávy pro trvanlivé subskripce bez registrovaného odběratele do doby, než bude mít opět aktivního odběratele. Tuto situaci znázorňuje obrázek 10.



Obrázek 9: Odběratelé s běžnou subskripcí



Obrázek 10: Odběratelé s trvanlivou subskripcí

Trvanliví odběratelé se vytvářejí metodou Session.createDurableSubscription, jíž se musí kromě tématu, u něhož se má vytvořit trvanlivá subskripce, předat také název (identifikátor) trvalé subskripce. Následně vytvoření odběratelé se registrují u vytvořené subskripce právě prostřednictvím jejího názvu. Trvanlivou subskripci lze zrušit metodou Session.unsubscribe.

4.10.7 Transakce

Transakce představují základní jednotky práce, do nichž lze seskupit sérii operací (požadavků na odeslání či příjem zprávy), takže navenek vystupuje jako jediná operace, a tudíž proběhne buď celá úspěšně (pokud jsou úspěšně všechny její operace), anebo celá selže (když selže libovolná z operací).

ACID

JMS transakce dodržují sadu vlastností označovaných jako *ACID*, které garantují jejich spolehlivé zpracování. Termín ACID je zkratkou slov atomičnost (*atomicity*), konzistence (*consistency*), izolace (i*solation*) a trvanlivost (*durability*). Tyto vlastnosti spolu úzce souvisí, jedna podmiňuje druhou. Koncepce pojmu ACID vznikla v kontextu databázových transakcí, ale její význam je stejně tak platný i v doméně systémů pro zasílání zpráv.[18]

Atomičnost – Všechny operace přijetí či odeslání zpráv v rámci transakce musí být provedeny úspěšně, jinak nebude provedena žádná z nich, neboli pokud jedna část transakce selže, selže celá transakce. Transakce nemohou být tedy rozděleny na menší celky, ale musí být zpracovány celistvě.
Pro uživatele to znamená, že se nemusí obávat, že by se v důsledku selhání HW, sítě, systému, či samotné aplikace provedla jen část operací, což by uvedlo aplikaci do nekonzistentního stavu.

Konzistence – Tato vlastnost zajišťuje, že všechny zprávy přijaté či odeslané uvnitř transakce zůstávají v konzistentním stavu. Přesněji řečeno to znamená, že každá transakce převádí aplikaci z jednoho konzistentního stavu do jiného konzistentního stavu.
Pokud je z nějakého důvodu prováděna transakce, která porušuje pravidla konzistence, je celá transakce odvolána a aplikace je obnovena do původního konzistentního stavu.

Izolace – Požadavek na to, aby v průběhu provádění transakce nemohly jiné transakce přistupovat k týmž datům. Ačkoliv může uvnitř systému běžet současně více transakcí, každá transakce musí běžet bez ohledu na jiné transakce.
Výjimkou je případ, kdy jedna transakce vyžaduje tatáž data, jaká jiná transakce současně modifikuje. V takové situaci musí tedy první transakce počkat na skončení druhé transakce a až poté může přistoupit k požadovaným datům. Nesplněním tohoto principu by se aplikace mohla ocitnout v nekonzistentním stavu. Tato vlastnost v podstatě znamená, že se žádné dvě transakce nesmí vzájemně ovlivňovat.

Trvanlivost – Tato vlastnost zajišťuje, že když je transakce jednou potvrzena, tak všechny změny, která způsobila budou zachovány, a tím i schopnost obnovy jednou potvrzených změn i pro případ jakéhokoliv druhu selhání systému.

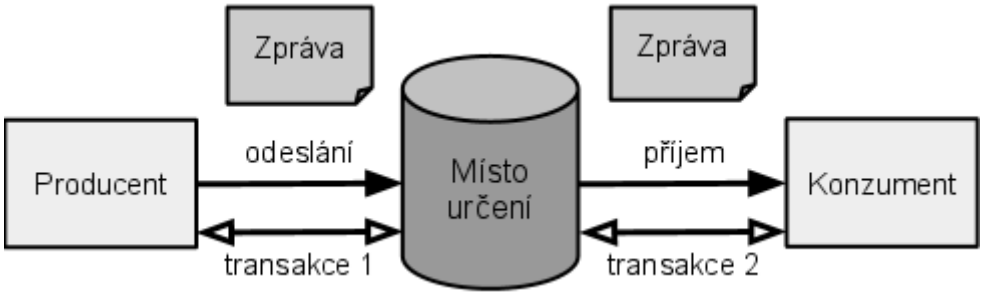
Lokální JMS transakce

Transakce jsou v JMS API podporovány prostřednictvím objektů relací (tj. rozhraním Session) jako jejich volitelná součást. JMS transakce seskupují vyprodukované a zkonzumované zprávy relace do série nedělitelných jednotek. Když je transakce potvrzena (metodou commit) je potvrzeno přijetí všech zkonzumovaných zpráv a všechny vyprodukované zprávy jsou odeslány. Naopak pokud je transakce odvolána (metodou rollback), jsou vyprodukované zprávy zničeny a zkonzumované zprávy jsou obnoveny a opětovně doručeny.

Proces obnovy navrátí relaci do stavu, jenž zaujímala po poslední potvrzené transakci, přičemž se pokusí znovu doručit všechny ještě stále platné zprávy (tzn. že množina odesílaných zpráv se může lišit od původní).

Při kombinaci několika požadavků na odeslání a příjem zpráv uvnitř jediné transakce je nutné vzít v úvahu pořadí těchto operací. Bezproblémové jsou případy, kdy se transakce skládá výhradně z požadavků na odeslání, nebo pouze z požadavků na příjem zpráv. Bezpečná je také situace, kdy požadavky na příjem zpráv předchází požadavkům na odeslání. Problémy vyvstanou při použití modelu požadavek-odpověď, tedy když v rámci jedné transakce předchází požadavek na odeslání zprávy požadavku na přijetí zprávy odpovědi na odeslaný požadavek. Při takovém pořadí operací dojde totiž k tzv. uváznutí aplikace. Odeslání zpráv požadavku totiž neproběhne dokud není transakce potvrzena, to se však nestane, dokud není přijata zpráva odpovědi na požadavek. *Transakce nesmí obsahovat žádné požadavky na příjem zpráv, které závisejí na odeslání jiných zpráv v téže transakci.*[9]

Důležité je také podotknout, že produkce a konzumace téže zprávy nemůže být součástí jedné JMS transakce. Producent a klient spolu nekomunikují přímo, pouze prostřednictvím poskytovatele. Odeslání a příjem zprávy tak reprezentují dvě oddělené sady interakcí s prostředkovatelem, které mohou být součástí vlastní transakce, nikoliv však společné (viz obrázek 11).



Obrázek 11: Použití lokálních transakcí JMS API

Distribuované transakce

JMS specifikace nevyžaduje, aby poskytovatel podporoval distribuované transakce. Nicméně pokud tomu tak je, může JMS poskytovatel vystupovat v roli monitoru distribuovaných transakcí, přičemž řízení transakcí provádí prostřednictvím JTA.

Ačkoliv je možné, aby si JMS klienti řídili distribuované transakce sami, není tento způsob doporučován. Podpora JTA je cílená především pro poskytovatele určené k integraci do Java EE aplikačního serveru. Nejsnáze lze tedy využít distribuovaných transakcí implementováním JMS klientů jako komponent EJB (více viz kapitoly 4.3 a 4.11.3).

4.11 Použití JMS API v Java EE aplikacích

Platforma Java EE nabízí nové případy užití pro předávání zpráv, a tedy i využití JMS API. Nové možnosti představují primárně distribuované komponenty EJB. Použití JMS API v Java EE aplikacích je v mnoha ohledech podobné jeho použití v samostatných klientských aplikacích. Hlavní rozdíly leží především v používání administrovaných objektů, transakcí a správě zdrojů.[9]

Neboť životní cyklus EJB komponent je plně řízen EJB kontejnerem, a pod kontrolou programátora není tedy jejich tvorba, ani destrukce, a navíc jsou určeny mnohonásobnému volání, tak není vhodné vyhledávat JNDI službou administrované objekty či vytvářet objekty spojení a relací pokaždé, je-li daná EJB komponenta potřeba. Lepším přístupem je udržovat tyto objekty po celou dobu existence EJB komponenty. Nesmí se však zapomínat na uvolnění vyhrazených zdrojů, nejsou-li již dále potřeba. Se specifikací platformy Java EE verze 5 a novými prvky jazyka Java je možno využít pro získávání administrovaných objektů anotaci `@Resource`, která zajistí, že požadovaný objekt bude aplikačním serverem lokalizován a přiřazen do specifikované proměnné. Klient tak nemusí vůbec používat JNDI API.

Největší rozdíl leží v používání transakcí, neboťnamísto lokálních transakcí využívají EJB komponenty distribuované transakce spravované samotným EJB kontejnerem. JMS nevylučuje ani možnost použití JTA pro vlastní (uživatelské) řízení transakcí.

4.11.1 Session Beans

Relační EJB mohou být v Java EE aplikaci použity k produkci či synchronní konzumaci zpráv. Protože však blokující synchronní příjem zadržuje systémové prostředky serveru, není synchronní příjem zpráv (metodou `receive`) zrovna dobrou programátorskou technikou pro komponenty EJB. Vhodnější je využít časovaného synchronního příjmu nebo příjmu asynchronního prostřednictvím zprávami řízených bean.

4.11.2 Message Driven Beans

Technologie EJB od verze 2.0 obsahuje speciální typ komponent označovaný jako zprávami řízené beany, neboli *english Message Driven Beans* (*MDB*), které umožňují Java EE aplikacím zpracovávat JMS zprávy asynchronně.

Hlavním rozdílem MDB oproti relačním beanám je, že MDB nemají žádná lokální či vzdálená rozhraní, a nemohou tak být volány klientem přímo, nýbrž pouze prostřednictvím přijaté zprávy zaslané klientem na místo určení, na němž MDB poslouchá.

MDB je komponenta, která vystupuje jako posluchač zpráv a může spolehlivě konzumovat zprávy z front zpráv nebo témat (prostřednictvím trvanlivých subskripcí). Zprávy mohou být zaslány jakoukoliv Java EE komponentou (aplikačním klientem, jinou EJB komponentou nebo webovou komponentou), nebo i aplikací či systémem, který vůbec nepoužívá Java EE technologii.

MDB, podobně jako posluchač zpráv, implementuje rozhraní MessageListener (navíc k rozhraní javax.ejb.MessageDrivenBean na platformě J2EE, resp. je deklarována s anotací @MessageDriven na platformě JavaEE) a jeho metodu onMessage, která je automaticky zavolána při příchodu nové zprávy. Od posluchačů zpráv v klasických aplikacích se však liší v několika ohledech spojených především s jejich umístěním a během v EJB kontejneru, který automaticky vykonává některé úlohy (tvorba instancí, tvorba konzumentů zpráv a registrace posluchačů apod.).

4.11.3 Distribuované transakce

Java EE aplikace běžně používají distribuované transakce za účelem zajistit integritu přístupů k externím zdrojům. V Java EE aplikaci, která používá JMS API, je možno distribuovaných transakcí využít pro kombinování požadavků na odeslání či příjem zprávy s přístupy do databáze či dalšími operacemi nad externími zdroji.

Distribuované transakce mohou být dvojího typu:

- *kontejnerem řízené transakce* (*container-managed transactions*, zkráceně *CMT*) — EJB kontejner sám řídí průběh transakcí bez nutnosti je explicitně potvrzovat či odvolávat. Toto je doporučovaný typ transakcí pro Java EE aplikace používající JMS API
- *beanou řízené transakce* (*bean-managed transactions*, ve zkratce *BMT*) -– Jak název napovídá, při tomto typu transakcí leží zodpovědnost za jejich řízení na implementaci bean samotných, přičemž hranice transakcí se vyznačují explicitně prostřednictvím JTA.

Využitím beanou řízených transakcí lze zpracovávat zprávu, či její část ve více než jedné transakci, nebo zcela mimo transakční kontext. Ve většině případů však nabízejí kontejnerem řízené transakce větší spolehlivost a pohodlí, a jsou proto vhodnější.

4.11.4 Webové komponenty

Specifikace platformy Java EE nedefinuje, jak implementují JMS API webové komponenty (tj. komponenty využívající Java Servlet API nebo technologii JavaServer Pages). Nicméně webová komponenta může (podobně jako komponenta Session Bean) zprávy odesílat a synchronně konzumovat, avšak nesmí je konzumovat asynchronně. Kvůli zadržování prostředků serveru je opět vhodnějším způsobem konzumace zpráv synchronní časovaný příjem, než příjem blokující.

5 Zabezpečení systémů předávání zpráv

Předpokladem úspěšného nasazení a rozšíření messaging systémů (a obecně jakékoliv technologie) v komerční oblasti je zajištění jejich bezpečnosti. Aplikace totiž mnohdy zacházejí s citlivými daty, která nesmí padnout do nepovolaných rukou (např. rodná čísla, čísla účtů apod.).

Produkty z oblasti Message-Oriented Middleware nabízejí rozličné způsoby řešení zabezpečení svých dat nejen proti případným útokům, ale také ochrany před jejich ztrátou v důsledku systémového selhání. Za data jsou v doméně messaging systémů považovány přenášené zprávy.

Systémy předávání zpráv vyžadují zabezpečení hned na několika místech. Předně je nutno zabezpečit komunikaci (tj. komunikační kanály) mezi klienty a messaging serverem, tzn. zajistit integritu a utajení přenášených zpráv, takže je nebude možno smysluplně odposlouchávat a falšovat. Zároveň je nutné zařídit, aby k messaging serveru přistupovali pouze ověření a oprávnění klienti, tj. nedovolit cizím klientům využívat messaging systém. Tutéž ochranu je nutné zajistit i pro jednotlivé fronty a témata zpráv, tedy aby k nim mohly přistupovat (zasílat a číst zprávy) jen určité aplikace.

Mnohdy je nutné zabezpečit vlastní data obsažená ve zprávách před samotnými aplikacemi, neboť vzhledem k anonymitě klientských aplikací a neexistenci těsné vazby mezi nimi, tedy situaci, kdy odesílatel zprávy nezná jejího příjemce, a tedy si nemůže být zcela jist, že se citlivé informace obsažené ve zprávách nedostanou k někomu nepovolanému. Stejně tak by zase příjemce měl mít jistotu, že jim obdrženou zprávu opravdu odeslal očekávaný klient. V některých případech může být navíc požadováno, aby odesílatel zprávy nemohl popřít její odeslání.

Důležitá je také otázka zabezpečení dat pro případ jakéhokoliv selhání, tj. selhání hardware (např. porucha pevného disku), sítě (např. ztráta spojení v průběhu komunikace), systému (např. nedostatek paměti a jiných zdrojů), či samotné aplikace (vlivem programátorské chyby nebo jiné neočekávané události). Aplikace předávající kritická data (např. bankovní transakce) si obvykle nemohou dovolit ztratit ani jedinou zprávu.

5.1 Obecné bezpečnostní principy

Bezpečnost je široký pojem, pokrývající úkoly pro zabezpečení dat tak, aby k nim měly přístup pouze oprávněné osoby, aby byla chráněna před neoprávněnou manipulací, poškozením, či prozrazením, aby byla bezpečně uchovávána a přenášena, aby se předešlo jejich podvržení a nešlo popřít jejich vytvoření.

Bezpečnostní funkce se obvykle dělí do pěti skupin podle bezpečnostního principu, jenž zajišťují:[19]

- *autentizace* (*authentication*) – zajištění, že komunikující strany jsou ty, za které se vydávají (zajištění identity)
- *řízení přístupu* (*access control*) – ochrana před neautorizovaným využitím systémových prostředků

- *utajení dat* (*data confidentiality*) – ochrana před neautorizovaným odhalením (prozrazením) dat
- *integrita dat* (*data integrity*) – ochrana přenášených dat proti neautorizované modifikaci
- *nepopiratelnost zodpovědnosti* (*non-repudiation*) – znemožnění popření odeslání, či příjmu zprávy

Mezi mechanizmy používané k implementaci těchto služeb patří:

- kryptografická ochrana důvěrnosti (šifrování dat)
- kryptografická ochrana integrity (hašovací funkce)
- systémy řízení přístupu
- kryptografická autentizace
- elektronický (digitální) podpis
- řízení přenosů zpráv
- zapouzdření zpráv
- ověřování třetí stranou (notářské služby)

5.2 Zabezpečení komunikačních kanálů

Zabezpečení všech výše uvedených principů se dá sice zajistit na aplikační vrstvě (tj. v samotné aplikaci), ale tento přístup klade velké nároky na vývojáře, kteří by museli sami implementovat veškeré bezpečnostní mechanismy. Proto je výhodnější a z hlediska použití transparentnější implementovat zabezpečení alespoň některých principů již na nějaké nižší úrovni.

5.2.1 Zabezpečení na transportní vrstvě

Obvykle se komunikace zabezpečuje na transportní vrstvě přidáním mezivrstvy do komunikace v podobě protokolu *SSL* (*Secure Sockets Layer*), či jeho následníka *TLS* (*Trasport Layer Security*). Oba poskytují autentizaci komunikujících stran prostřednictvím výměny certifikátů, načež následně dochází k zašifrování přenášených dat, což zajišťuje integritu a utajení dat. Dochází-li k přenosu dat za pomoci protokolu HTTP, tak po předřazení vrstvy SSL/TLS mluvíme již o HTTPS. Tyto protokoly však nepodporují nepopiratelnost, která je důležitá zvláště u komerčních systémů. Nepopiratelnost zodpovědnosti se pak obvykle zabezpečuje připojením digitálního podpisu k odeslané zprávě.

5.2.2 Zabezpečení na síťové vrstvě

Podobné bezpečnostní mechanismy lze do komunikace zařadit dokonce již na síťové vrstvě pomocí rozšíření protokolu IP označovaného jako *IPSec*. Jedná se o sadu bezpečnostních protokolů a algoritmů poskytující opět autentizaci, integritu a utajení dat prostřednictvím jejich šifrování. IPSec je základním stavebním prvkem, na nichž je postaven princip zakládání tzv. *virtuálních privátních sítí* (*VPN*), tzn. vytváření uzavřených,

soukromých a důvěryhodných síťových infrastruktur prostřednictvím veřejných a nedůvěryhodných počítačových sítí (Internet).

5.2.3 Zabezpečení na aplikační vrstvě

Těmito způsoby jsou zprávy chráněny, když jsou přenášeny médiem, ne však už v úložištích na pevném disku. Tento požadavek se zabezpečuje až na aplikační vrstvě, kdy je třeba šifrovat vlastní obsah přenášené zprávy. Tento přístup vyžaduje spolupráci odesílatele a příjemce, neboť oba musí znát použitý šifrovací algoritmus a především šifrovací klíč.

Častým formátem obsahu předávaných zpráv je XML. Pro zabezpečení takových zpráv na aplikační úrovni lze využít některé technologie a standardy, vyvinuté konzorciem W3C přímo k tomuto účelu. *XML Signature* zaručuje autenticitu, integritu dat a jejich nepopiratelnost. Jedná se o způsob, jak k XML dokumentům připojit digitální podpisy, které mohou podepisovat různé jeho části, jež vytvořily různé strany. Stejně tak technologie *XML Encryption* byla vytvořena, aby s její pomocí bylo možno zašifrovat pouze některé elementy XML dokumentů a utajit tak jejich obsah před neautorizovanými stranami.[20]

5.3 Řízení přístupu

Pro omezení přístupu k messaging systémům, respektive serverům, na nichž běží, se běžně používají nástroje (SW i HW) souhrnně označované jako *firewally*, které bývají často implementovány jako tzv. *paketové filtry*. Ty na základě tzv. seznamů řízení přístupu ACL (*Access Control List*) filtrují pakety příchozí a odchozí komunikace, přičemž zahazují pakety, které nevyhovují nastaveným podmínkám, resp. propustí pouze ty pakety, které vyhovují. Při filtrování mohou být zohledňovány předchozí pakety (tj. komunikační kontext – tzv. stavové filtry), či nikoliv, takže se bere v potaz pouze obsah aktuálního paketu (bezestavové filtry).

Na podobném principu mohou být založeny i mechanismy omezující přístup k jednotlivým frontám a tématům zpráv (např. HornetQ umožňuje definovat uživatelské role a sadu povolených operací nad frontami pro tyto role).

5.4 Zabezpečení uchování dat

Většina messaging produktů využívá pro ukládání a uchovávání zpráv externí databázové systémy. Proto pro zabezpečení takových dat platí postupy definované dodavatelem databáze. Taková bezpečnost je obvykle závislá na podpoře transakčního zpracování a zálohování databáze. Přístup k jednotlivým tabulkám databáze je pak obvykle řízen opět na základě rolí, v nichž se uživatel nachází.

Některé produkty nabízejí vlastní způsob ukládání a uchovávání zpráv v tzv. žurnálech, tj. souborech s vlastní správou. Tyto systémy (např. HornetQ) pak většinou neumožňují přistupovat k těmto souborům, a tedy frontám a tématům v nich obsažených, žádné cizí aplikaci.

5.5 Bezpečnostní principy JMS

Specifikace Java Message Service sama nedefinuje žádná bezpečnostní pravidla pro přenášení a uchovávání zpráv. Vše je ponecháno na implementaci daného messaging serveru. JMS specifikace však definuje některé postupy a mechanismy, které zajišťují spolehlivý přenos zpráv od klienta na server a opačným směrem tak, aby nebyla žádná zpráva ztracena, pokud na tom aplikaci záleží. Těmito prostředky tak dovoluje nastavit aplikacím potřebnou úroveň spolehlivosti a výkonu.

JMS stanovuje způsob deklarace transakčního zpracování, tedy odesílání a přijímání zpráv v rámci nedělitelných jednotek práce, které se musí provést celé úspěšně, jinak budou v případě selhání byť jediné operace všechny odvolány a provedeny opětovně. Více o transakcích a obnově systému v případě selhání je uvedeno v kapitole 4.10.7.

Nejspolehlivějším způsobem přenosu důležitých zpráv je jejich produkce v perzistentním doručovacím režimu v rámci transakce a jejich konzumace opět uvnitř transakce buď z trvanlivé fronty (v PTP doméně) nebo prostřednictvím trvanlivé subskripce (v Pub/Sub doméně). Detaily o doručovacích režimech zpráv a trvanlivých subskripcích jsou rozepsány v kapitolách 4.10.3 a 4.10.6.

JMS specifikace však upozorňuje, že definované doručování zpráv právě a pouze jednou (*once-and-only-once delivery*) v sobě nezahrnuje zničení zpráv v důsledku vypršení jejích platností či dalších administrativních kritérií (např. ztráty v důsledku omezení zdrojů). Konfigurace adekvátních zdrojů a výpočetní síly pro JMS aplikace je v kompetenci administrátorů, kteří musí být dostatečně obeznámeni s prostředky použitého JMS poskytovatele zajišťujícími spolehlivost přenosu zpráv.[1]

Na platformě Java lze navíc využít pro řešení autentizace klientů a jejich autorizace k provádění různých operací bezpečnostní framework *Java Authentication and Authorization Service* (*JAAS*), který poskytuje prostředky pro jednotnou tvorbu přihlašovacích modulů, čímž umožňuje separaci záležitostí týkajících se uživatelské autentizace od další aplikační logiky.

6 HornetQ

Příkladem softwaru typu Message Oriented Middleware je open source projekt HornetQ divize JBoss společnosti Red Hat, Inc. Tento projekt byl původně vyvíjen pod názvem JBoss Messaging jako jeho verze 2.0 a je určen pro integraci do aplikačního serveru JBoss 6.0 jako výchozí implementace JMS poskytovatele. Je napsán kompletně v jazyce Java, takže dokáže běžet na jakékoliv platformě s běhovým prostředím Javy. Právě tento produkt byl použit pro implementaci ukázkové aplikace.

HornetQ nabízí[15] široké spektrum vlastností a schopností. Na jejich kompletní soupis a obsáhlejší popis zde však není místo, stručně tak budou představeny jen jeho nejzajímavější vlastnosti. Zájemci naleznou podrobnější údaje v [5].

6.1 Základní vlastnosti

Ačkoliv HornetQ implementuje specifikaci JMS verze 1.1 v plné míře, a tedy podporuje veškeré její součásti, tak jeho samotné jádro je navrženo zcela bez vazby na JMS. Java Message Service tak vystupuje pouze jako jakási fasáda na klientské straně nad aplikačním rozhraním jádra a nabízí tak alternativu k možnosti použití vlastního nativního API HornetQ, které je samozřejmě optimalizováno tak, aby bylo možno s jeho pomocí využít HornetQ v jeho plné síle.

Obecně lze říci, že HornetQ bylo navrženo s ohledem na možnost použití více různých komunikačních protokolů a API. Od verze 2.1 podporuje HornetQ protokol STOMP, takže vystupuje jako STOMP broker a dokáže jeho prostřednictvím komunikovat s klienty vytvořenými na jiných platformách než je Java. Do budoucna se navíc také počítá s podporou protokolu AMQP či ReSTful API. To by mimojiné znamenalo (a v současné době je to již prostřednictvím protokolu STOMP možné), že zprávy mohou na HornetQ server od klientů jedním protokolem přicházet a z něj odcházet k jiným klientům jiným protokolem (viz obrázek 12).

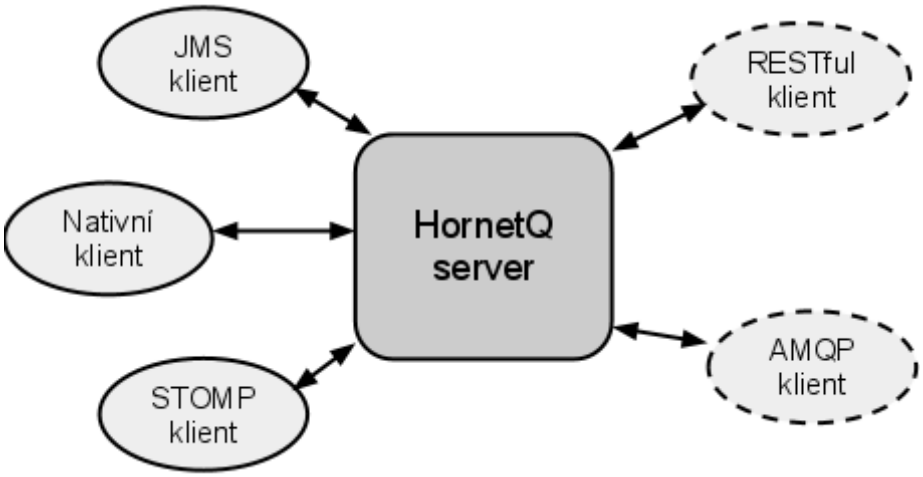
Pro přenos zpráv na síťové vrstvě používá HornetQ knihovnu Netty, což je mimochodem jeho jediná závislost na knihovně třetí strany. Kromě komunikace v rámci jednoho virtuálního stroje Javy, tak HornetQ nabízí i komunikaci prostřednictvím TCP, SSL, HTTP a HTTPS (pro případ, že bezpečnostní politika a firewally nedovolují jinou komunikaci), či dokonce možnost využití servletu jako zástupce, který přesměrovává požadavky na HornetQ server na strojích, kde již běžný webový server běží.

6.2 Koncepce jádra

Vývojáři, kteří se nechtějí zabývat záludnostmi JMS či potřebují využít některou specifickou funkci HornetQ, mají možnost využít nativní HornetQ API. Jeho koncepce v některých ohledech podobná JMS, v jiných se liší. Obecně lze říci, že je především jednodušší, neboť odstraňuje rozdíly mezi frontami, tématy a subskripcemi.

V HornetQ jsou entitami, kam jsou zprávy zasílány, či z nichž jsou odebírány, *adresy*. Ke každé adrese může být připojeno libovolné množství *front*. Když na server dorazí zpráva

[15] Poslední dostupnou stabilní verzí byla k 30. červnu 2010 verze 2.1.1.



Obrázek 12: HornetQ server a klienti

je směrována do všech front připojených k cílové adrese. Fronta navíc může definovat filtr zpráv, které ji mají být směrovány. Tento návrh poskytuje velkou variabilitu. JMS frontu lze tedy modelovat jako jednu adresu, k níž je připojena jediná HornetQ fronta. Implementace JMS tématu zahrnuje adresu, k níž je připojeno mnoho front, kde každá reprezentuje jednu subskripci.

Fronty mohou být trvanlivé, tedy takové, že zprávy v nich obsažené přetrvají pád serveru, netrvanlivé (zprávy nepřežijí restart serveru), či dočasné, které jsou automaticky odstraněny při uzavření spojení. Fronty mohou být připojeny pouze k jediné adrese.

HornetQ navíc umožňuje vytvářet hierarchie adres a podle nich směrovat zprávy s využitím zástupných symbolů. Konzumenti zpráv tak mohou odebírat zprávy nejen z konkrétní adresy, ale z více adres. Odběratel může být vytvořen pro adresu news.cz.$*$ a následně odebírat zprávy publikované pro adresy news.cz.politics, news.cz.sport apod. Těchto hierarchií je navíc využito pro specifikaci různých nastavení pro různé větve adres.

Kontextem pro produkci a konzumaci zpráv jsou objekty ClientSession, které zapouzdřují jak samotné spojení s messaging serverem, tak relaci mezi ním a klientem. Zároveň podporuje transakční zpracování a to včetně distribuovaných JTA transakcí.

6.3 Integrace

Ačkoliv je HornetQ zamýšlen jako výchozí JMS poskytovatel pro aplikační server JBoss, neomezuje se jeho nasazení pouze na tento případ. Poskytuje totiž vlastní plně funkční JCA adaptér, a tedy může být snadno integrován do jakéhokoliv Java EE aplikačního serveru. Při tomto typu integrace tak může poté využít veškeré prostředky, které mu Java EE platforma nabízí, například tedy zasílání zpráv z komponent EJB, či servletů a jejich příjem prostřednictvím komponent Message Driven Bean. Komunikace s Java EE aplikačním serverem pak probíhá plně prostřednictvím JCA adaptéru a nikoliv přímo.

Neboť je HornetQ vytvořen jako sada jednoduchých Java objektů (tzv. *POJO*) téměř bez závislostí na knihovnách třetích stran může být integrován do aplikací vyžadujících předávání zpráv interně bez nutnosti jej vystavovat jako messaging server. Lze jej instanciovat, konfigurovat a spustit programově přímo v kódu aplikace a nikoliv na úrovni systému.

Pro případy, kdy systém vyžaduje pouze předávání zpráv bez potřeby další funkcionality Java EE platformy, může být HornetQ nasazen jako nezávislý samostatně běžící server. Toto nastavení zahrnuje vlastní messaging server, JMS službu a službu JNDI spolu s aplikačním rozhraním pro správu všech prostředků. Tento typ nasazení HornetQ interně používá jako běhové prostředí pro POJO objekty JBoss Microcontainer.

6.4 Perzistence

HornetQ se nespoléhá jako mnohé jiné MOM produkty na ukládání zpráv do relačních databází, ale pro uchovávání zpráv používá vlastní vysoce výkonný žurnál, optimalizovaný pro požadavky spojené s ukládáním zpráv.

Žurnál je tvořen sadou souborů, které má plně ve vlastní správě. Pro potřeby optimalizace (minimalizace operací s náhodným přístupem) se jednotlivé záznamy připojují vždy na konec žurnálu. Jednotlivé záznamy jsou tak tvořeny operacemi, které zprávy přidávají, aktualizují, či mažou. Žurnál se dělí do třech částí, kde jedna spravuje vazby mezi adresami a frontami, jiná uchovává veškeré informace související s JMS a třetí uchovává samotné zprávy.

Přístup k vlastnímu souborovému systému je abstrahován tak, že umožňuje různé implementace. V současné době jsou tak poskytovány dvě implementace. První používá rozhraní Java NIO (standardní balíček `java.nio` platformy J2SE), které nabízí velice dobrý výkon. Pro operační systémy Linux je navíc dostupná implementace používající knihovnu pro asynchronní vstup a výstup (tzv. *Asynchronous I/O*, neboli *AIO*). Tato implementace nabízí dokonce vyšší výkonnost než Java NIO.

HornetQ navíc umožňuje spolehlivě přenášet zprávy obrovského objemu při současné podpoře transakcí, a to dokonce na strojích s malou operační pamětí. Tvůrci uvádějí[5], že otestovali odeslání a příjem zpráv až do velikosti 8 GB. Teoreticky by však mělo být možné přenášet zprávy až do velikosti $2^{63} - 1$ B, limitem je tedy velikost volného prostoru na pevném disku. Takovéto zprávy se neukládají do žurnálu, ale samostatně.

6.5 Bezpečnost

Zabezpečení front realizuje HornetQ zavedením uživatelských rolí a k nim přiřazených práv. Práva zahrnují tvorbu a odstraňování trvanlivých i netrvanlivých front, možnost zasílání a příjmu zpráv. Tato práva mohou být nastavena pro fronty individuálně, či hromadně s využitím adres se zástupnými symboly.

Pro autentizaci a autorizaci klientů lze však využít i službu JAAS a HornetQ server tak snadno zapojit do již existující bezpečnostní infrastruktury a spravovat tak uživatelské účty centrálně pomocí prostředků JAAS.

6.6 Distribuované předávání zpráv

HornetQ obsahuje mnohé prostředky pro podporu tvorby rozsáhlých distribuovaných systémů. Umožňuje propojování více messaging serverů a následným směrováním zpráv zajistí rovnoměrné rozložení zátěže na všechny servery.

6.6.1 Clustery

HornetQ umožňuje vytvářet skupiny HornetQ serverů (tzv. *clustery*). Každý uzel (*node*) clusteru je HornetQ server zpracovávající vlastní zprávy a spojení. Avšak zprávy jsou mezi těmito uzly směrovány tak, aby se co možná nejvíce vyrovnalo zatížení serverů, tedy aby jeden uzel nestíhal zprávy zpracovávat, zatímco jiný „odpočíval“. Automatickou distribucí zpráv mezi uzly clusteru podle aktuální zátěže a počtu aktivních konzumentů se tak předchází možnému hladovění či přetížení jednotlivých uzlů.

Uzly v clusteru mohou být propojeny na základě mnoha různých topologií. Nejčastějšími jsou symetrická a řetězová topologie. Při prvním z nich jsou jednotlivé uzly navzájem propojeny, tj. každý s každým, takže každý uzel ví o každé frontě a aktivních konzumentech všech ostatních uzlů. U řetězové topologie jsou uzly propojeny sériově vždy (s výjimkou krajních uzlů) s předchozím a následujícím uzlem. Zprávy jsou poté propagovány řetězem až k potřebnému uzlu.

6.6.2 Vysoká dostupnost

S provozem více uzlů zapojených do jednoho clusteru souvisí i vlastnost označovaná jako vysoká dostupnost (*High Availability*). Jedná se o schopnost systému pokračovat v činnosti i při selhání jednoho či více serverů.[5] Tato vlastnost, která v sobě zahrnuje i schopnost klientských spojení plynule přecházet mezi servery při výpadku (*failover*) tak, aby klient mohl nadále fungovat, je tedy dosti klíčová v distribuovaném a paralelním prostředí.

HornetQ umožňuje vytvářet dvojice serverů, kde jeden vystupuje jako aktivní (živý) server a druhý jako záložní, který je neaktivní, dokud nedojde k výpadku živého serveru. Tyto servery mohou buďto sdílet jeden žurnál, nebo může při selhání jednoho dojít k replikaci jeho žurnálu na záložní server.

6.6.3 Přemostění

Tvorba clusterů a záložních serverů je umožněna prostřednictvím tzv. přemostění (*bridge*). Ty obvykle fungují tak, že konzumují zprávy z fronty na jednom serveru a přeposílají je do jiné fronty na jiném serveru. Tyto servery však nemusí vůbec náležet k jednomu clusteru, ba co více HornetQ dovoluje vytvářet přemostění na libovolné jiné JMS messaging servery, tedy nikoliv nezbytně HornetQ servery.

V případě nasazení v nespolehlivých prostředích (např. ve WAN) se pří ztrátě spojení pokouší o opětovné přípojení, dokud není cíl opět dostupný, čímž je zajištěno i spolehlivé doručování zpráv. Přemostění mohou být navíc konfigurovány tak, že přeposílají pouze zprávy vyhovující definovanému filtru.

7 Implementace ukázkové aplikace

Součástí práce je i implementace ukázkového aplikace demonstrující použití rozličných prvků Java Message Service API a konceptů předávání zpráv. Jedná se o systém, který simuluje chod firmy zabývající se výrobou a distribucí nábytku. Systém sestává z několika komponent a aplikací standardní i enterprise edice platformy Java. Navíc je i jeden z klientů implementován v jazyce C# pro platformu .NET. Tento klient slouží k demonstraci schopnosti komunikovat se serverem HornetQ protokolem STOMP napříč platformami.

K realizaci systému byl použit HornetQ server ve verzi 2.1.1, který byl nasazen v rámci aplikačního serveru JBoss 5.1. Potřebu ukládat data zajistila integrovaná databáze HyperSQL (HSQLDB) ve verzi 1.8. Pro implementaci STOMP klienta byla použita .NET knihovna NMS verze 1.3.0.

K samotnému vývoji systému bylo použito integrované vývojové prostředí Eclipse 3.5 Galileo (ve variantě pro Java EE vývojáře) spolu se sadou zásuvných modulů JBoss AS Tools. To vše na platformě Java SE 6. STOMP klient v jazyce C# byl pak vyvíjen v IDE Microsoft Visual Studio 2005, Standard Edition.

7.1 Popis domény

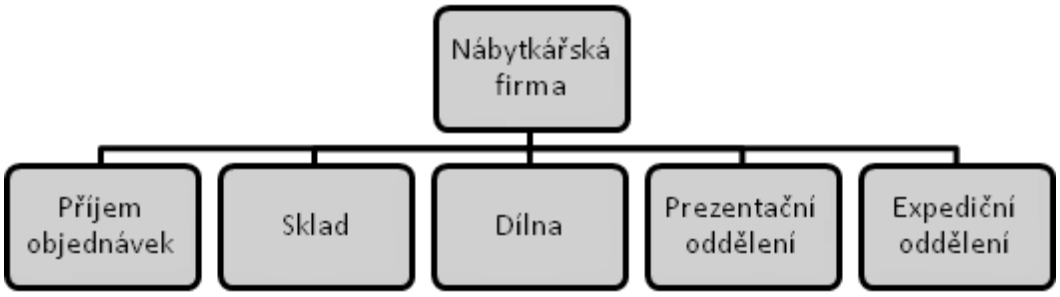
Jak již bylo řečeno, implementovaný systém simuluje chod nábytkářské firmy. Konkrétně pak proces objednávání nábytku, který zahrnuje i jeho výrobu a distribuci, a proces nového typu nábytku. Cílem však nebylo co nejvěrnější zachycení skutečného fungování a reálných potřeb takového podniku. Záměrem bylo především demonstrovat široké spektrum možností JMS.

7.1.1 Struktura firmy

Nábytkářská firma se skládá z několika spolupracujících oddělení, kde každé z nich má jiné kompetence (viz obrázek 13).

- Oddělení příjmu objednávek – přijímá a zpracovává objednávky, informuje distributora o dostupnosti zboží
- Sklad – spravuje skladování zboží, přijímá (naskladňuje) nový nábytek a vydává objednaný nábytek
- Prezentační oddělení – spravuje katalog nabízeného zboží
- Dílna – vyrábí nábytek, komunikuje s dodavateli materiálu
- Expediční oddělení – zajišťuje dodání nábytku distributorům

Kromě těchto oddělení firma (resp. oddělení dílny) dále spolupracuje s podniky, kteří jí dodávají materiál potřebný k výrobě nábytku. Firma neprodává svůj nábytek přímo zákazníkům, ale má své distributory, kteří obstarávají tyto záležitosti. Někteří distributoři se mohou navíc specializovat jen na konkrétní typu nábytku.



Obrázek 13: Organizační struktura nábytkářské firmy

7.1.2 Firemní procesy

Pro účely ukázkové aplikace byly z mnoha procesů probíhajících v takové nábytkářské firmě vybrány především dva hlavní procesy. Prvním z nich je zpracování objednávek a druhým je vývoj nového nábytku. Ostatní procesy (např. učetnictví) mohou sice tvořit dokonce významnější složku chodu firmy, avšak pro demonstraci použití JMS nejsou nezbytné.

Zpracování objednávky

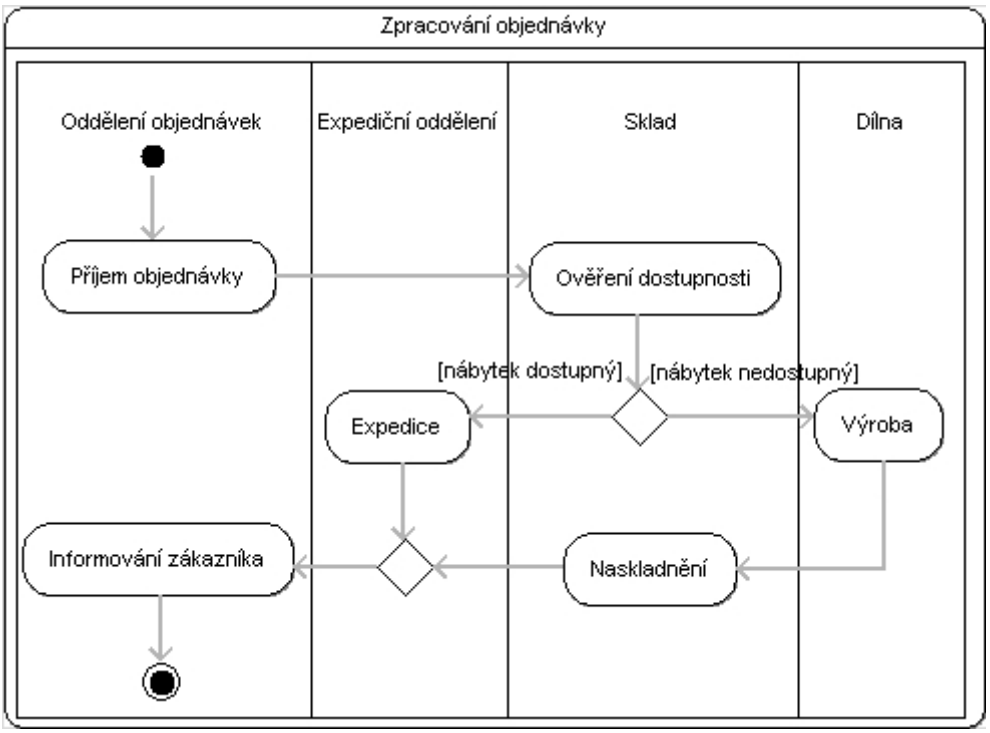
Proces zpracování objednávky (viz aktivitní diagram na obrázku 14) začíná, jakmile přijde od některého distributora objednávka na dodání nábytku, tak ji zpracuje oddělení příjmu objednávek. Tato činnost spočívá v ověření dostupnosti objednávaného zboží na skladě, jeho vyhrazení a následném zaúkolování expedičního oddělení požadavkem dodání dostupného nábytku distributorovi, který je následně vyrozuměn o situaci.

Sklad v případě, že objednávaný nábytek není dostupný v požadovaném množství, či zásoba nábytku klesne pod určité minimum objedná výrobu daného nábytku na dílně. Samotný podproces výroby nábytku (viz diagram na obrázku 15) sestává z paralelního zajištění potřebného materiálu od dodavatelů a výroby různých součástí. Poté jsou jednotlivé komponenty smontovány do výsledného kusu nábytku.

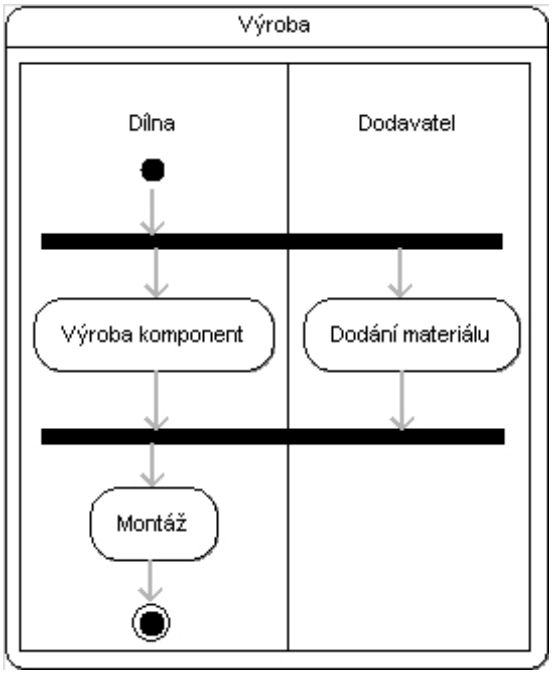
Jakmile je výroba nábytku hotova, je tento předán skladu k naskladnění, přičemž je o této události informováno oddělení objednávek, které informuje distributora o dostupnosti nábytku.

Vývoj nového nábytku

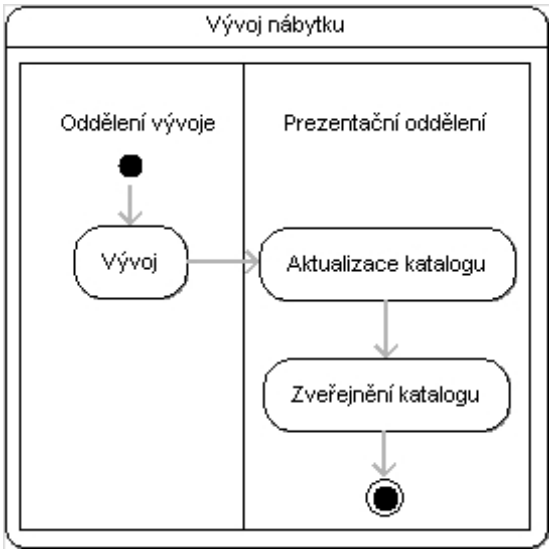
Nezávisle na této činnosti funguje proces vývoje nábytku (obrázek 16). Oddělení vývoje pracuje podle nějakých zadání na vývoji nových druhů nábytku. Jakmile je nějaký nový druh nábytku vyvinut, je o tomto informováno prezentační oddělení, kde je novému nábytku přiřazen název, popis a evidenční číslo, načež je tento nábytek zařazen do katalogu. Poté tento aktualizovaný katalog uveřejní, takže distributoři, kteří daný typ nábytku distribuují, mohou aktualizovat své portfolio.



Obrázek 14: Aktivitní diagram procesu zpracování objednávky



Obrázek 15: Aktivitní diagram procesu výroby nábytku



Obrázek 16: Aktivitní diagram procesu vývoje nábytku

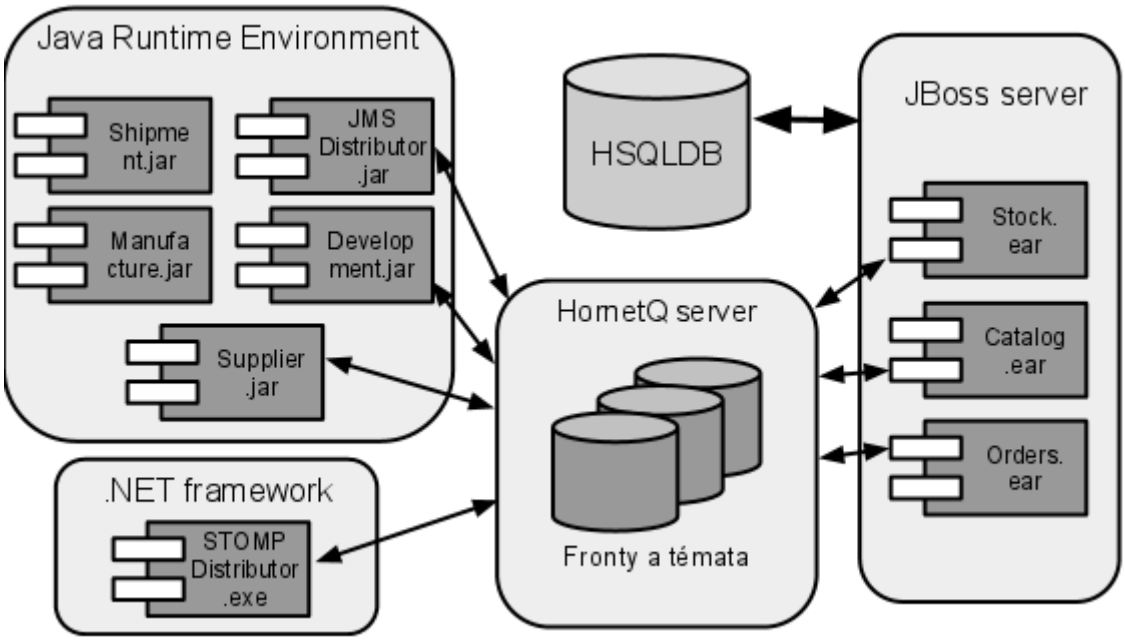
7.2 Implementace systému

Každý účastník výše zmíněných procesů, tj. každé oddělení, je implementována jako samostatná aplikace. Některé jsou však představovány klasickými konzolovými desktopovými aplikacemi (distribuovány jako Java archívy – přípona jar), jiné zase Java enterprise aplikacemi (přípona ear), určenými pro běh v rámci Java EE aplikačního serveru (v mém případě JBoss). Aplikace distributora implementovaná pro platformu .NET (přípona exe) je opět desktopovou aplikací, navíc s grafickým uživatelským rozhraním. Centrálním bodem je tak messaging server HornetQ, který obhospodařuje fronty a témata zpráv, stejně jako příjem, směrování a rozesílání zpráv klientům. Situaci znázorňuje diagram 17.

Vlastní komponenty jsou implementovány především s důrazem na prezentaci různých prostředků JMS API. Takže třeba zprávy předávané mezi komponentami jsou pokaždé různého typu (použity jsou jak textové, tak proudové, mapové i objektové zprávy). Jejich samotné chování se kromě provedení nezbytných akcí pro zajištění chodu systému omezuje jen na výpisy hlášek o průběhu zpracovávání zpráv.

Aplikace distributorů (tj. JMSDistributor.jar a STOMPDistributor.ear) komunikují se systémem vždy prostřednictvím textových zpráv, obsahujících data ve formě XML dokumentu, což je formát, jehož obsah lze zpracovat na všech platformách.

Hlavním rozdílem mezi komponentami běžícími v rámci aplikačního serveru JBoss a těmi běžícími jako samostatné aplikace leží v práci s administrovanými objekty. Zatímco klasické aplikace získávají tyto objekty prostřednictvím služby JNDI, tak komponenty běžící uvnitř EJB kontejneru obdrží tyto objekty na základě příslušných anotací pomocí vstřikování zdrojů (dependency injection), což poněkud zvětšuje pohodlí práce (např. viz výpis 2 na straně 38).



Obrázek 17: Rozmístění komponent a jejich komunikace

Aplikace běžící nad aplikačním serverem JBoss používají k asynchronní konzumaci zpráv zprávami řízené beany (MDB). Vzhledem k tomu, že o životní cyklus komponent MDB se stará EJB kontejner, je nutné jejich konfiguraci provést deklarativně opět prostřednictvím anotací. Příklad takové konfigurace lze vidět na výpisu 10, který ukazuje deklaraci MDB jako trvanlivého odběratele z tématu zpráv.

```
@MessageDriven(activationConfig = {
        @ActivationConfigProperty(propertyName = ”destinationType”, propertyValue = ”javax.jms.
            Topic”),
        @ActivationConfigProperty(propertyName = ”destination”, propertyValue = ”topic/manufacture
            /newFurniture”),
        @ActivationConfigProperty(propertyName = ”clientID”, propertyValue = ”orders.mdb.
            NewFurnitureManufacturedProcessor”),
        @ActivationConfigProperty(propertyName = ”subscriptionDurability”, propertyValue = ”Durable
            ”),
        @ActivationConfigProperty(propertyName = ”subscriptionName”, propertyValue = ”
            NewFurnitureManufacturedProcessor”) })
@ResourceAdapter(”hornetq−ra.rar”)
public class NewFurnitureManufacturedProcessor implements MessageListener {...}
```

Výpis 10: Nastavení message driven beany pomocí anotací

Nejdůležitější anotací je @MessageDriven, která označuje že daná třída je MDB a které se pomocí anotací @ActivationConfigProperty nastavují nejrůznější hodnoty jako název a typ místa určení, z níž má MDB zprávy konzumovat (destination a destinationType), nebo

údaje nutné k vytvoření trvanlivé subskripce (clientID atd.). Anotace @ResourceAdapter je specifickou anotací aplikačního serveru JBoss, kterého instruuje, že daný MDB nepoužívá standardní JMS adaptér JBossu, ale ten dodávaný s HornetQ. Za povšimnutí rovněž stojí, že třída MDB implementuje rozhraní MessageListener.

7.2.1 Místa určení systému

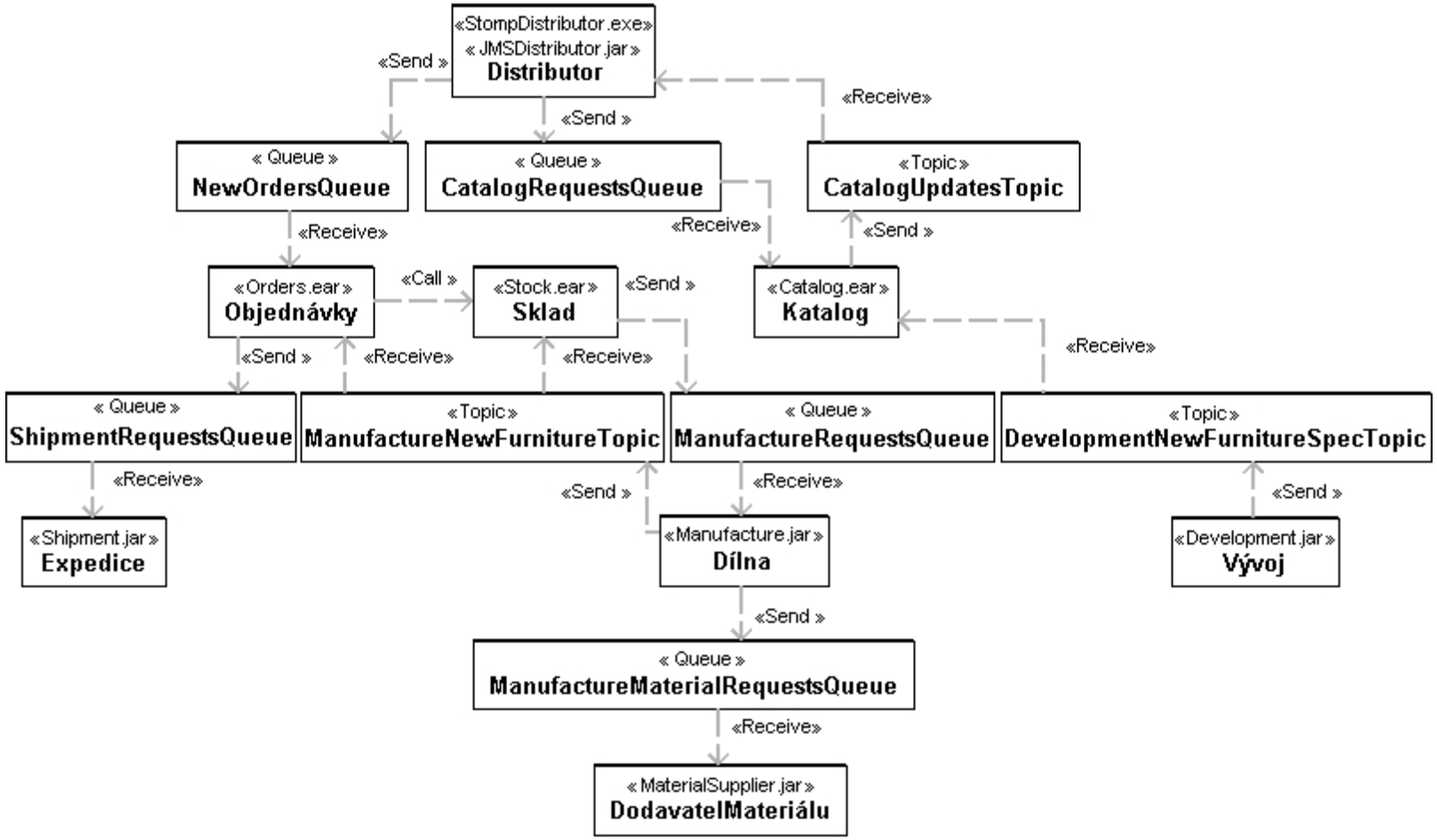
Komunikační rozhraní mezi jednotlivými komponentami a aplikacemi systému tvoří fronty a témata JMS zpráv (s výjimkou komponenty obsluhující požadavky na sklad StockRequestsBean, která je implementována jako bezestavová session bean, a je tedy volána synchronně prostřednictvím svého business rozhraní).

- NewOrdersQueue – fronta zpráv, kam distributor zasílá své objednávky na nábytek a odkud si je vyzvedává oddělení příjmu objednávek
- ShipmentRequestsQueue – fronta zpráv, kam oddělení příjmu objednávek zasílá své požadavky na dodání zboží a odkud si je vyzvedává expediční oddělení
- ManufactureRequestsQueue – fronta zpráv, kam oddělení skladu zasílá své požadavky na výrobu nábytku a odkud si je vyzvedává dílna
- ManufactureMaterialRequestsQueue – fronta zpráv, kam dílna zasílá své požadavky na dodání materiálu potřebného k výrobě nábytku a odkud si je vyzvedávají jednotliví dodavatelé
- ManufactureNewFurnitureTopic – téma zpráv, kam dílna zveřejňuje oznámení o vyrobení nábytku a odkud je odebírají oddělení příjmu objednávek a skladu
- CatalogRequestsQueue – fronta zpráv, kam distributor zasílá své požadavky na vydání katalogu nábytku a odkud si je vyzvedává prezentační oddělení
- DevelopmentNewFurnitureSpecTopic – téma zpráv, kam oddělení vývoje zveřejňuje oznámení o vyvinutí nového druhu nábytku a odkud je odebírá prezentační oddělení (katalog)
- DevelopmentNewFurnitureSpecTopic – téma zpráv, kam oddělení vývoje zveřejňuje oznámení o vyvinutí nového druhu nábytku a odkud je odebírá prezentační oddělení (katalog)
- CatalogUpdatesTopic – téma zpráv, kam prezentační oddělení zveřejňuje oznámení o výskytu nového nábytku v katalogu a odkud je odebírají distributoři nábytku

Místa určení vystupující mezi jednotlivými odděleními lze vidět na obrázku 18.

7.2.2 Komponenty systému

Jednotlivé aplikace systému sestávají z několika komponent. Enterprise aplikace se skládají z bezestavových session bean (SLSB) a message driven bean (MDB), zatímco ty desktopové většinou z třídy vlastní aplikace a posluchače zpráv.



Obrázek 18: Místa určení mezi komunikujícími odděleními

Vztahy a komunikaci mezi komponentami prostřednictvím definovaných míst určení zachycují diagramy na obrázcích 19 a 20. Dočasné fronty (stereotyp TemporaryQueue) mají sice v diagramech specifikován název, avšak ve skutečnosti jej není třeba.[16]

Následující odstavce stručně představují jednotlivé aplikace a komponenty. Podrobnější informace o jejich implementaci jsou zaznamenány v programátorské dokumentaci.

Oddělení příjmu objednávek – Orders.ear

Agendu příjmu objednávek zajišťuje aplikace Orders.ear, resp. její třídy umístěné v podbalíčcích balíčku orders. Jedná se o:

- OrdersBean – SLSB pro přístup k datům vztaženým k agendě zpracování objednávek
- OrderProcessor – MDB zpracovávající nové objednávky distributorů (textová zpráva s XML obsahem) a vydávající jim odpovědi o dostupnosti nábytku (opět XML textová zpráva) na principu request/reply, zároveň vytváří požadavky na dodání nábytku (objektová zpráva)
- NewFurnitureManufacturedProcessor – MDB reagující na událost vyrobení nábytku (proudová zpráva) odesláním informativního emailu (pouze symbolicky) distributorům, kteří na tento nábytek čekají

Expediční oddělení – Shipment.jar

Aplikace Shipment.jar obstarává agendu expedice nábytku distributorům. Její kód se nachází v balíčku shipment.

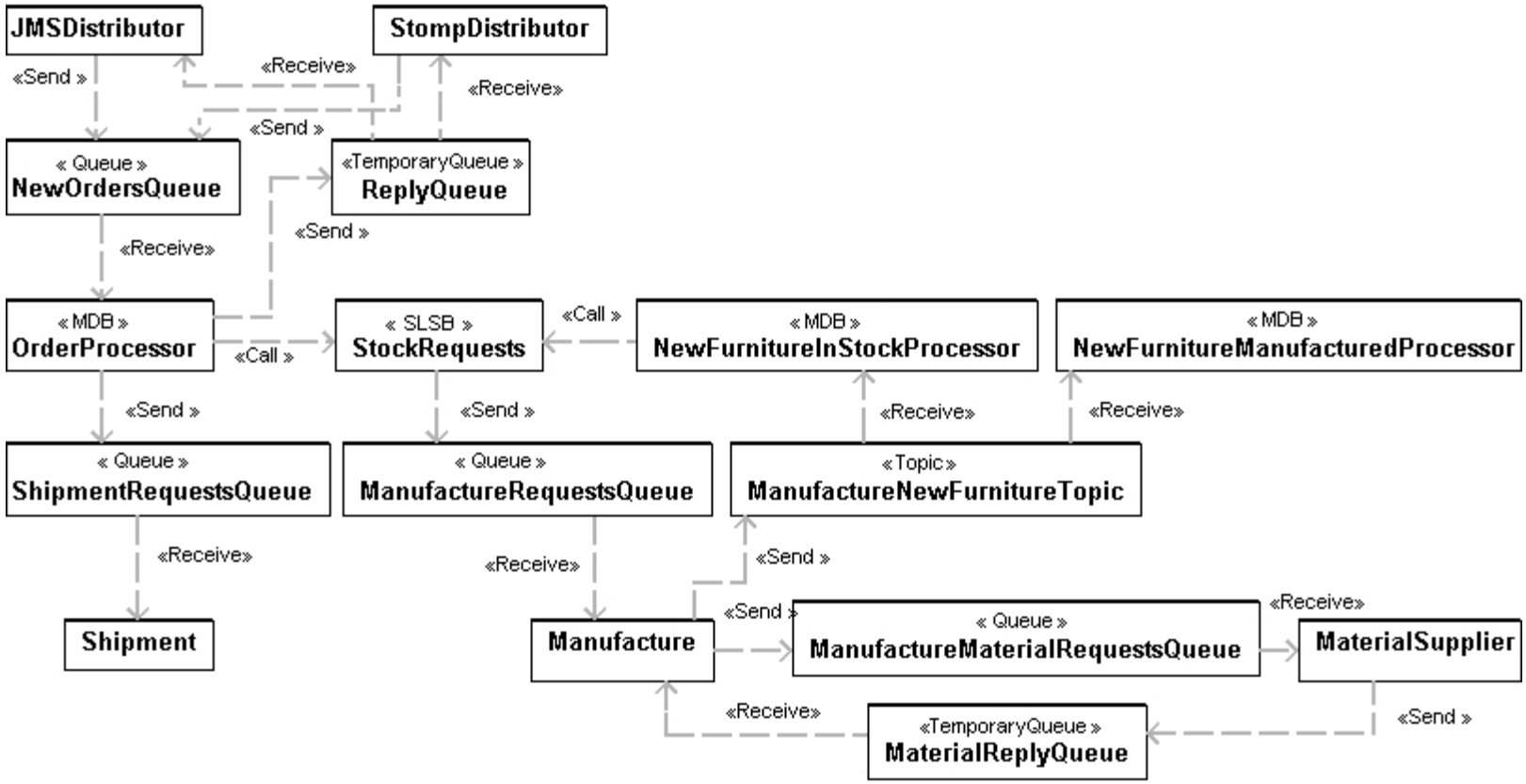
- Shipment – třída aplikace expedičního oddělení, registruje posluchače zpráv pro asynchronní příjem požadavků
- ShipmentRequestListener – posluchač zpracovávající požadavky (objektová zpráva) na dodání zboží (pouze symbolicky)

Oddělení kladu – Stock.ear

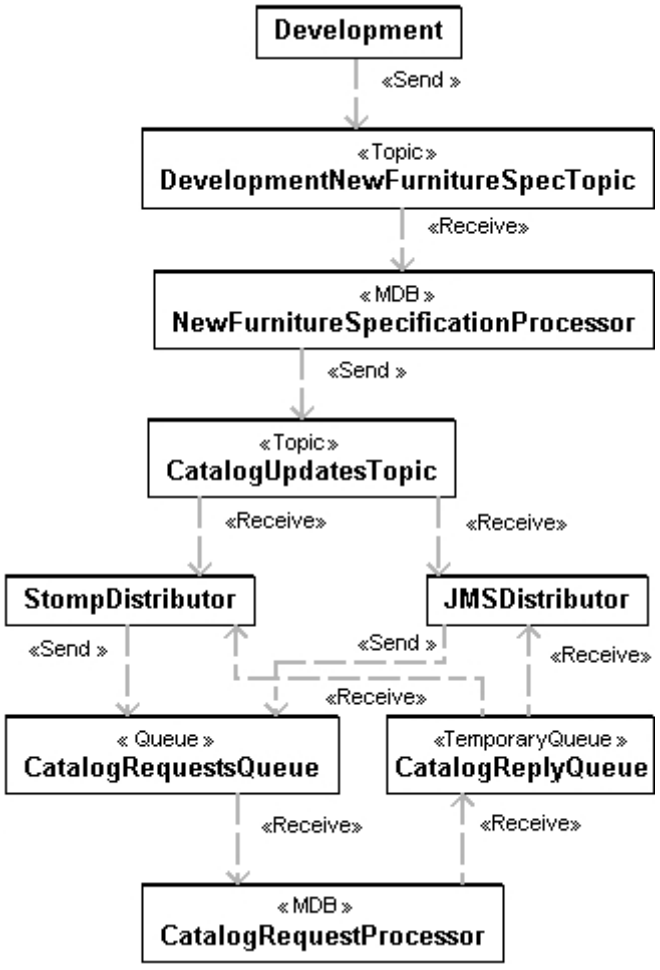
Chod skladu zabezpečuje enterprise aplikace Stock.ear, konkrétně třídy nacházející se v podbalíčcích balíčku stock.

- StockBean – SLSB pro přístup k datům evidovaných skladem
- StockRequestsBean – SLSB obsluhující požadavky na poskytnutí nábytku, ověření jeho dostupnosti, či příjmu nového nábytku na sklad; při poklesu množství nábytku pod určitou mez objedná na dílně výrobu nového (mapová zpráva různé priority)
- NewFurnitureInStockProcessor – MDB zpracovávající událost výroby nového nábytku, zajišťující jeho naskladnění

[16] Neboť protokol STOMP nepodporuje princip dočasných destinací, tak je pro účely distributora nábytku, který jej využívá vytvořena speciální fronta ReplyQueue, kam jsou odesílány veškeré odpovědi na jeho požadavky.



Obrázek 19: Vztahy komponent v procesu zpracování objednávky



Obrázek 20: Vztahy komponent v procesu vývoje nábytku

Oddělení dílny – Manufacture.jar

Dílna je reprezentována aplikací Manufacture.jar a třídou manufacture.Manufacture, která registruje posluchače zpráv ManufactureRequestListener, jenž zpracovává požadavky na výrobu nábytku.

Výroba nábytku spočívá v zajištění materiálu potřebného k jeho výrobě vydáním požadavků na dodání materiálu (textová zpráva) a následného časovaného synchronního příjmu odpovědi reprezentující dodávaný materiál. Po vyrobení požadovaného množství nábytku je o tomto uvědoměn sklad a oddělení objednávek (využití Pub/Sub modelu).

Dodavatelé materiálu – MaterialSupplier.jar

MaterialSupplier.jar reprezentuje aplikaci dodavatele materiálu, který čeká na požadavky na dodání materiálu (textová zpráva), který po jejich přijetí dodá odesílateli (opět textová zpráva), přičemž je využito filtrování zpráv podle typu materiálu.

Oddělení vývoje – Development.jar

Proces vývoje nových druhů nábytku je představován třídou Development v balíčku development, kdy v pravidelných minutových intervalech (s 50% pravděpodobností) vyvine náhodně nový druh nábytku a odešle jeho specifikaci (objektová zpráva) prezentačnímu oddělení.

Prezentační oddělení – Catalog.ear

Agendu správy katalogu nábytku zajišťuje enterprise aplikace Catalog.ear a třídy balíčku catalog.

- CatalogBean – SLSB pro přístup k datům katalogu nábytku
- NewFurnitureSpecificationProcessor – MDB zpracovávající událost vyvinutí nového typu nábytku (objektová zpráva), aktualizuje příslušně katalog a zveřejňuje změny pro distributory (XML textová zpráva).
- CatalogRequestProcessor – MDB přijímající požadavky (XML textová zpráva) na výdej aktuálního katalogu nábytku, který vydává v podobě odpovědi (XML textová zpráva)

Distributoři – JMSDistributor.jar a StompDistributor.exe

Aplikace distributorů byly implementovány ve dvou variantách. První je konzolová Java aplikace využívající JMS, druhou je aplikace s grafickým rozhraním pro platformu .NET využívající protokol STOMP a knihovnu NMS.

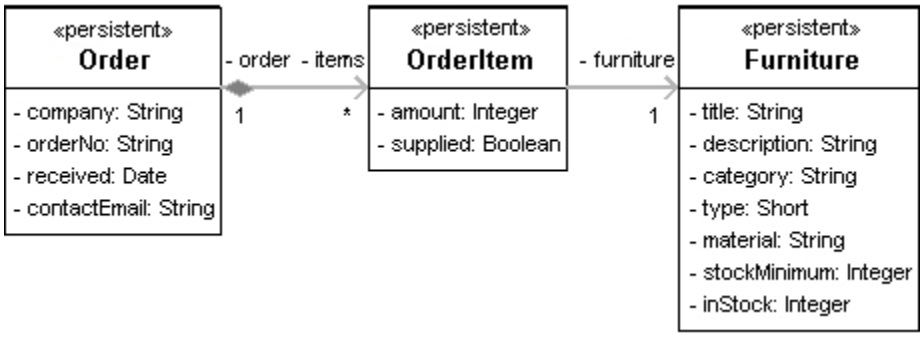
Obě aplikace potřebují ke svému běhu specifikovat několik údajů, jako je název společnosti, kontaktní email a dále kategorii a typ distribuovaného nábytku.

Aplikace si na začátku svého běhu vyžádá aktuální katalog a vytvoří posluchače aktualizací katalogu. Oba distributoři mají prostředky pro vytváření objednávek na nábytek. Bližší popis ovládání obou aplikací se nachází v uživatelské příručce.

7.2.3 Datová vrstva

Data systému (nábytek, objednávky apod.) jsou uchovávána v relační databázi. K této databázi systém přistupuje prostřednictvím bezestavových session bean, které reprezentují DAO objekty. Ty jsou volány při zpracovávání zpráv message driven beanami. Tímto je tedy zároveň demonstrováno fungování distribuovaných transakcí, takže selže zpracování zprávy, budou odvolány i všechny změny provedené v databázi v téže transakci. Tyto DAO objekty jsou představovány rozhraními Catalog, Stock, Orders a jejich implementačními třídami CatalogBean, StockBean a OrdersBean.

Samotné schéma databáze bylo navrženo co nejjednodušší, aby vyhovovalo potřebám systému a zároveň zbytečně nekomplikovalo přístup k datům. Báze dat je tvořena třemi tabulkami: FURNITURE, FORDER a ORDER_ITEM schématu STOCK. Pro každou tabulku existuje třída (definovaná v balíčku common.model), jejíž instance reprezentují jeden záznam dané tabulky. Jedná se tak o jednoduchý druh objektově-relačního mapování. Tyto třídy a vazby mezi nimi, které zároveň vyjadřují vazby mezi tabulkami, lze vidět na třídním diagramu na obrázku 21.



Obrázek 21: Třídy databázového schématu systému

8 Shrnutí a závěr

Primární oblastí, kde předávání zpráv hraje klíčovou roli, je integrace heterogenních systémů. Ať už z důvodu strukturálních změn, nových obchodních záměrů, či prostě přechodu na novou technologii, stále více a více společností čelí problému, jak integrovat heterogenní systémy a aplikace uvnitř podniku, či mezi nimi. Není neobvyklé setkat se s mnoha různými technologiemi a platformami v jediné společnosti.

Předávání zpráv nabízí schopnost zpracovávat požadavky asynchronně, čímž lze redukovat či úplně odstranit úzká místa systému, a zvýšit tak produktivitu koncových uživatelů a celkovou škálovatelnost systému. Vzhledem k tomu, že předávání zpráv potlačuje vazbu mezi komponentami, poskytuje architektura systému používajícího předávání zpráv vysoký stupeň pružnosti.[10]

Podnikové messaging systémy umožňují dvěma a více aplikacím vyměňovat si informace ve formě zpráv, kde zpráva je jakýsi balíček obsahující důležitá data a hlavičky nutné pro směrování. Zprávy obvykle obsahují informace o nějaké podnikové transakci či informují aplikace o výskytu nějaké události.

Pomocí middleware produktu založeného na zasílání zpráv jsou zprávy přenášeny od jedné aplikace k druhé, přičemž messaging systém zajistí správnou distribuci zpráv mezi aplikacemi a navíc často poskytuje další prostředky pro spolehlivou výměnu velkého množství zpráv, jako jsou vyvažování zátěže, škálovatelnost a podpora transakcí.

Výměna zpráv neprobíhá přímo mezi aplikacemi, nýbrž prostřednictvím virtuálních kanálů zvaných místa určení, kam jsou zprávy adresovány. Jakákoliv aplikace, která registruje svůj zájem o určité místo určení, může z něj přijímat zprávy. Tímto je potlačena vazba mezi aplikacemi odesílajícími a přijímajícími zprávy.

Messaging systémy používají různé formáty zpráv a protokoly pro jejich výměnu, základní sémantika je však shodná. MOM produkty také poskytují aplikační rozhraní pro tvorbu zpráv, plnění daty, přiřazení směrovacích informací a jejich odeslání stejně jako pro jejich příjem.

Podobnost principů messaging produktů dává prostor pro vznik aplikačních rozhraní nezávislých na messaging systému. Takovým API je i Java Message Service, takže JMS aplikace může posílat a přijímat zprávy od různých messaging systémů kompatibilních s JMS.

Předávání zpráv má své místo především v rozsáhlých podnikových systémech, kde je třeba zajistit spolupráci více subsystémů za současného zachování vzájemné nezávislosti. Přesto však nelze jeho princip uplatnit za všech okolností, neboť asynchronní způsob předávání zpráv nemůže uspokojit veškeré požadavky na takovou integraci.

9 Literatura

[1] Hapner, M., et al.: *Java Message Service Specification, v. 1.1* [online], Sun Microsystem, Inc., 2002. Dostupné z WWW: `http://java.sun.com/products/jms/docs.html`

[2] Modi, T.: *Practical Java Message Service*, Manning Publications Company, 2002, ISBN: 1930110138.

[3] *Middleware*, Wikipedia, the free encyclopedia [online]. Publikováno v červnu 2002, citováno v květnu 2010. Dostupné z WWW: `http://en.wikipedia.org/wiki/Middleware`

[4] *Remote procedure call*, Wikipedia, the free encyclopedia [online]. Publikováno v únoru 2002, citováno v květnu 2010. Dostupné z WWW: `http://en.wikipedia.org/wiki/Remote_procedure_call`

[5] *HornetQ 2.1 User Manual* [online], Red Hat, Inc., 2010. Dostupné z WWW: `http://hornetq.sourceforge.net/docs/hornetq-2.1.1.Final/user-manual/en/pdf/HornetQ_UserManual.pdf`

[6] *Message-oriented middleware*, Wikipedia, the free encyclopedia [online]. Publikováno v lednu 2004, citováno v červnu 2010. Dostupné z WWW: `http://en.wikipedia.org/wiki/Message_Oriented_Middleware`

[7] *Enterprise service bus*, Wikipedia, the free encyclopedia [online]. Publikováno v srpnu 2004, citováno v červnu 2010. Dostupné z WWW: `http://en.wikipedia.org/wiki/Enterprise_service_bus`

[8] Ranck, D.: *Web services vs. a message queue system*, SearchSOA.com [online]. Citováno v červnu 2010. Dostupné z WWW: `http://searchsoa.techtarget.com/expert/KnowledgebaseAnswer/0,289625,sid26_gci818509,00.html`

[9] Haase, K.: *Java Message Service API Tutorial* [online], Sun Microsystems, Inc., 2002. Dostupné z WWW: `http://download.oracle.com/javaee/1.3/jms/tutorial/jms_tutorial-1_3_1.pdf`

[10] Richards, M.; Monson-Haefel, R.; Chappell, D. A.: *Java Message Service, 2nd Edition*, O'Reilly Media, Inc., 2009, ISBN: 978-0-596-52204-9.

[11] *.NET Messaging API*, The Apache Software Foundation, domovské stránky projektu [online]. Citováno v červnu 2010. Dostupné z WWW: `http://activemq.apache.org/nms/index.html`

[12] *Representational State Transfer*, Wikipedia, the free encyclopedia [online]. Publikováno v srpnu 2004, citováno v červnu 2010. Dostupné z WWW: `http://en.wikipedia.org/wiki/REST`

[13] *Advanced Message Queuing Protocol*, Wikipedia, the free encyclopedia [online]. Publikováno v červnu 2006, citováno v červnu 2010. Dostupné z WWW: http://en.wikipedia.org/wiki/AMQP

[14] *The Stomp Project*, Codehaus.org, domovské stránky projektu [online]. Citováno v červnu 2010. Dostupné z WWW: http://stomp.codehaus.org/Home

[15] *Streaming Text Oriented Messaging Protocol*, Wikipedia, the free encyclopedia [online]. Publikováno v únoru 2007, citováno v červnu 2010. Dostupné z WWW: http://en.wikipedia.org/wiki/Streaming_Text_Orientated_Messaging_Protocol

[16] *Java EE version history*, Wikipedia, the free encyclopedia [online]. Publikováno v březnu 2007, citováno v červnu 2010. Dostupné z WWW: http://en.wikipedia.org/wiki/Java_EE_version_history

[17] *Java Message Service*, Wikipedia, the free encyclopedia [online]. Publikováno v březnu 2002, citováno v červnu 2010. Dostupné z WWW: http://en.wikipedia.org/wiki/Java_Message_Service

[18] *ACID*, Wikipedia, the free encyclopedia [online]. Publikováno v červenci 2002, citováno v červenci 2010. Dostupné z WWW: http://en.wikipedia.org/wiki/ACID

[19] Kunderová, L.: *Bezpečnost IS/IT*, PEF MZLU v Brně, Ústav informatiky [online]. Citováno v červenci 2010. Dostupné z WWW: https://akela.mendelu.cz/~lidak/bis/index.htm

[20] Lasoň, M.: *Zabezpečení webových služeb*, VŠB-TU Ostrava, Fakulta elektrotechniky a informatiky, Katedra informatiky [online]. Citováno v červenci 2010. Dostupné z WWW: http://homel.vsb.cz/~las03/wss/wssecurity.pdf